

AURORA

Aurora 参考文档

Version: 1.0

目录

前言	v
1. 文档说明	v
2. 版权声明	v
1. Aurora 框架简介	1
1.1. 前言	1
1.2. SVN地址	3
2. Aurora框架概览	4
2.1. 面向配置, SOA, RESTful	4
2.2. 灵活的扩展机制	5
2.3. 固态的对象 vs 液态的数据	6
2.4. 接口抽象 vs 过程抽象	6
3. Aurora展示层(APL)	7
3.1. APL(Aurora Presentation Layer)概述	7
3.2. APL服务端工作原理	7
3.2.1. 基于组件重用的开发模型	7
3.2.2. 简单高效的组件模型	8
3.2.3. 无状态组件	10
3.2.4. 面向数据	10
3.2.5. 服务端组件 + 客户端组件	11
3.2.6. Aurora如何创建用户界面	12
3.2.7. 数据容器	12
3.2.8. 组件映射	12
3.2.9. BuildSession	12
3.2.10. 事件机制	12
3.2.11. 资源文件处理	12
3.2.12. 动态组件配置	12
3.3. APL UI组件	12
3.3.1. 核心Javascript API	12
3.3.2. DataSet	13
3.3.2.1. DataSet定义	13
3.3.2.2. 前言	14
3.3.2.3. 元数据(Metadata)	14
3.3.2.4. 校验(Validate)	15
3.3.2.5. 动态界面逻辑处理	16
3.3.2.6. DataSet常用操作与事件	16
3.3.3. 界面布局(Layout)	16
3.3.3.1. Box	16
3.3.3.2. VBox	17
3.3.3.3. HBox	18
3.3.3.4. Form	18
3.3.3.5. FieldSet	19
3.3.4. 编辑组件	19
3.3.4.1. TextField	19
3.3.4.2. NumberField	21

3.3.4.3. ComboBox	22
3.3.4.4. DateField	24
3.3.4.5. DatePicker	24
3.3.4.6. Lov	26
3.3.4.7. MultiLov	29
3.3.4.8. DateTimePicker	31
3.3.4.9. TextArea	32
3.3.4.10. Radio	33
3.3.4.11. CheckBox	35
3.3.5. Tab组件	37
3.3.5.1. Tab定义	37
3.3.5.2. Tab标签属性	38
3.3.6. Tree组件	38
3.3.6.1. Tab定义	38
3.3.6.2. 树节点渲染	39
3.3.6.3. Tree标签属性	40
3.3.7. Grid组件	40
3.3.7.1. 数据绑定	41
3.3.7.2. 列对齐	41
3.3.7.3. 列锁定	41
3.3.7.4. 调整列宽	42
3.3.7.5. 排序	42
3.3.7.6. 复合表头	42
3.3.7.7. 列渲染	43
3.3.7.8. 汇总列	43
3.3.7.9. 工具栏	44
3.3.7.10. 编辑器	45
3.3.8. Table组件	45
3.3.8.1. Table定义	45
3.3.8.2. Table与Grid的异同	46
3.3.9. 窗口组件(Window)	46
3.3.10. 上传组件	49
3.3.11. TreeGrid	49
3.4. 个性化及定制	51
3.4.1. 界面定制	51
3.4.2. 组件样式修改	51
3.4.3. 修改网页整体布局	51
3.5. 多语言支持	51
3.5.1. 基于数据库存储的多语言支持	51
3.5.2. 自定义多语言实现	51
3.5.3. Screen及模板资源文件中的多语言支持	51
4. Aurora服务层(ASL)	52
4.1. ASL(Aurora Service Layer)总体架构概述	52
4.2. 业务模型(Business Model)	52
4.2.1. BM的创建与使用	52
4.2.1.1. Business Model的基本属性	52
4.2.1.2. BM的使用	53
4.2.2. 通过BM执行查询	54
4.2.2.1. 字段, 别名与表达式	54

4.2.2.2. SQL Join	55
4.2.2.3. 查询	57
4.2.2.4. 自定义查询语句	61
4.2.3. 通过BM执行DML	62
4.2.3.1. 通过BM执行insert	63
4.2.3.2. 通过BM执行update	64
4.2.3.3. 通过BM执行删除	65
4.2.3.4. 批量操作	66
4.2.3.5. 级联操作	67
4.2.4. 自定义BM操作	68
4.2.4.1. 用自定义SQL实现CRUD操作	68
4.2.4.2. 存储过程的调用	69
4.2.5. 在BM中使用Feature	70
4.2.5.1. Feature示例: Standard Who	70
4.2.5.2. Aurora内建Feature介绍	72
4.2.6. BM继承	72
4.2.6.1. BM继承概述	72
4.2.6.2. 引用模式	72
4.2.6.3. 重载模式	74
4.2.6.4. 同名节点的属性处理	75
4.2.6.5. BM继承应用场合分析	75

前言

XXXXXX

1. 文档说明

各个章节相关负责人以及审校人员

表 1. Aurora 开发团队

序号	标题	撰写	Email
#1	APL总体架构概述	牛佳庆	njq.niu@hand-china.com
#2	DataSet	牛佳庆	njq.niu@hand-china.com
#3	界面布局	牛佳庆	njq.niu@hand-china.com
#4	Grid组件	牛佳庆	njq.niu@hand-china.com
#5	编辑组件	吴华真	huazhen.wu@hand-china.com

关于我们

Aurora 开发团队

隶属于上海汉得信息技术股份有限公司 (www.hand-china.com) MAS部门.

2. 版权声明

Aurora文档属于Aurora发行包的一部分，遵循LGPL协议。本翻译版本同样遵循LGPL协议。参与翻译的译者一致同意放弃除署名权外对本翻译版本的其它权利要求。

您可以自由链接、下载、传播此文档，或者放置在您的网站上，甚至作为产品的一部分发行。但前提是必须保证全文完整转载，包括完整的版权信息和作译者声明，并不能违反LGPL协议。这里“完整”的含义是，不能进行任何删除/增添/注解。若有删除/增添/注解，必须逐段明确声明那些部分并非本文档的一部分。

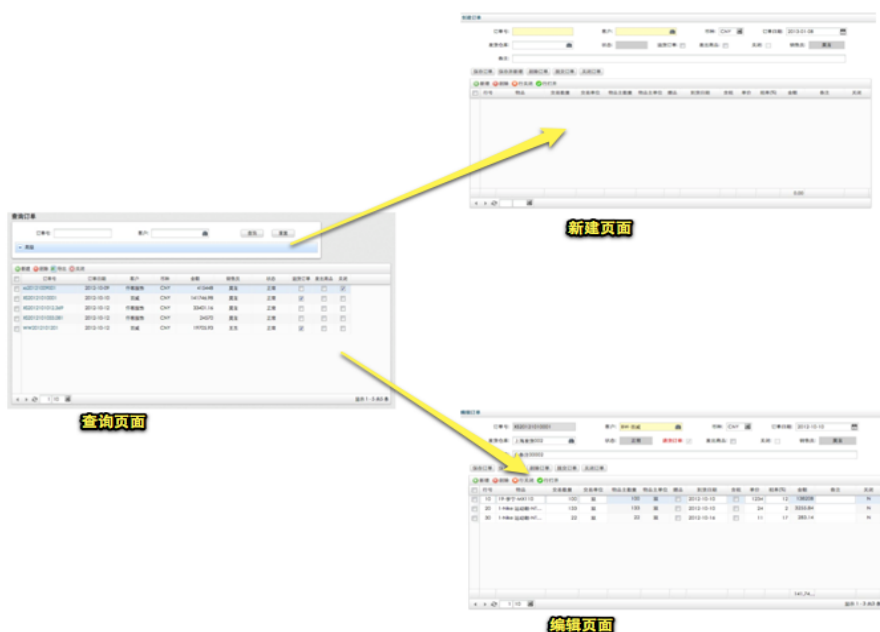
第 1 章 Aurora 框架简介

1.1. 前言

Aurora是一个基于J2EE的Web应用开发框架，主要面向企业应用。在介绍Aurora框架之前，我们先看看用Aurora开发出来的应用实例。鉴于企业应用通常都不会像J2EE宠物商店那么简单，我们将选择一个典型的业务：销售订单，来作为演示场景。我们的示例程序包括以下几个界面：销售订单录入，销售出库，销售订单查询。请点击[此处](#)查看在线演示程序：

...

从应用开发者的角度，Aurora与其它框架的最大区别，就是大多数功能都是通过XML配置文件，而不是Java代码来实现的。实际上，Aurora可以看做是一个配置解析器，通过解析配置文件来实现各种功能。我们以【销售订单】这个演示界面为例，说明在Aurora框架下应用程序是如何开发出来的。



首先我们来看前端界面部分。用户所访问的每一个界面（我们可以在demo系统中看到的.screen链接），都对一个XML格式的配置文件，它在server端被解析，产生HTML输出，生成供浏览器展示的UI。以...界面为例，按各种条件查询数据，点击某条记录能够在一个弹出窗口中编辑数据，像这样一个典型的企业应用界面，其源代码如下：

```
<a:screen xmlns:a="http://www.aurora-framework.org/application" trace="true">
  <a:init-procedure>
    ...
  </a:init-procedure>
  <a:view>
    <a:link id="ord_sales_order_create_screen" url=".....screen"/>
    <a:link id="lk_header_close" url="....svc"/>
    <script>...</script>
    <a:dataSets>
      <a:dataSet id="ord_order_status_ds" lookupCode="SALE_ORD_STATUS"/>
      ....
    </a:dataSets>
    <a:screenBody>
```

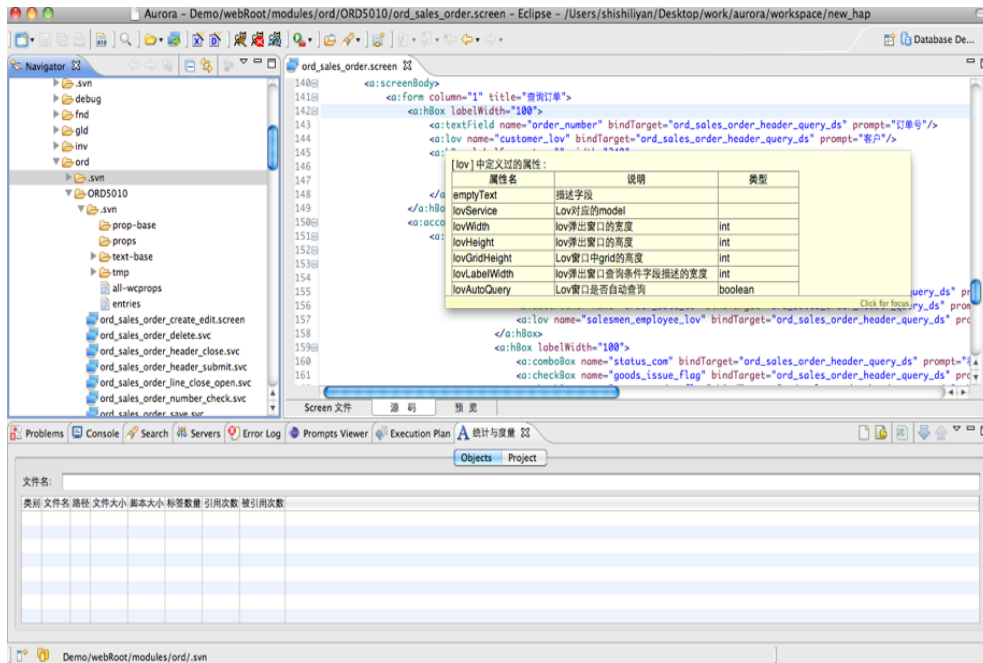
```

<a:form column="1" title="查询订单">
  <a:hBox labelWidth="100">
    <a:textField name="order_number" bindTarget="..." prompt="订单号"/>
    <a:lov name="customer_lov" bindTarget="..." prompt="客户"/>
    <a:hBox labelSeparator="" width="240">
      <a:button click="btn_query_order_header" prompt="" text="HAP_QUERY"/>
      <a:button click="btn_reset_query_ds" text="HAP_RESET"/>
    </a:hBox>
  </a:hBox>
  ...
</a:form>
<a:hBox/>
<a:grid bindTarget="ord_sales_order_headers_ds" height="350" navBar="true" width="1000">
  <a:columns>
    <a:column name="order_number" prompt="订单号" width="150"/>
    ...
    <a:column name="closed_flag" editor="e_cb" prompt="关闭" width="40"/>
  </a:columns>
  <a:editors>
    <a:checkBox id="e_cb"/>
  </a:editors>
  <a:toolBar>
    ...
    <a:button click="btn_close_order" icon="..." text="关闭"/>
  </a:toolBar>
</a:grid>
</a:screenBody>
</a:view>
</a:screen>

```

源文件在<init-procedure>部分说明如何获取该界面所需的数据（我们将在后面对其原理进行更详细的介绍），<view>部分说明整个界面是如何用基本的UI组件构建而成。例如，通过<dataset>组件，我们声明了在客户端端存储的数据集，以便实现主从数据联动、记录在客户端的新增修改删除、有效性校验等功能，并将它与后台提供实际数据存储服务的URL关联在一起；通过<grid>组件，我们以表格的形式将dataset中的数据展示出来，并提供编辑的功能；通过<lov>组件，我们可以提供对海量数据进行选择的组件，并支持auto-complete特性。

在上面的例子中，我们通过这几个配置文件，就可以完成一个典型的应用界面。虽然Aurora是基于J2EE的开发框架，但这个通过Aurora开发的应用程序没有一行Java代码。Aurora也提供了基于Eclipse的IDE，可对这些配置文件进行快速创建、可视化编辑，并提供语义有效性检查，能在很大程度上消除配置错误引发的问题。在IDE专题部分会有更详细的介绍。



1.2. SVN地址

Aurora的所有代码托管在Google Code。

- Uncertain

<https://aurora-project.googlecode.com/svn/trunk/uncertain>

- Aurora

<https://aurora-project.googlecode.com/svn/trunk/aurora>

- Aurora UI

<https://aurora-project.googlecode.com/svn/trunk/AuroraUI>

第 2 章 Aurora框架概览

2.1. 面向配置，SOA，RESTful

Aurora提供了一种面向配置的业务建模方式，能够快速实现数据库相关的操作，并以RESTful的风格将这些操作变为服务。

在示例程序中，对[订单]的查询及维护，就是通过这样一段配置来实现的：

```
<bm:model xmlns:bm="http://www.aurora-framework.org/schema/bm" xmlns:f="aurora.database.features" alias="h" baseTable="ord">
  <bm:fields>
    <bm:field name="order_number" databaseType="VARCHAR" datatype="java.lang.String"/>
    <bm:field name="return_order_flag" databaseType="VARCHAR" datatype="java.lang.String"/>
    ....
    <bm:field name="close_date" databaseType="DATETIME" datatype="java.sql.Date"/>
  </bm:fields>
  <bm:primary-key>
    <bm:pk-field name="sales_order_id"/>
  </bm:primary-key>
  <bm:data-filters>
    <bm:data-filter name="company_id" enforceOperations="query,update" expression="h.company_id=${/session/@company_id}/>
  </bm:data-filters>
  <bm:features>
    <f:standard-who/>
  </bm:features>
  <bm:query-fields>
    <bm:query-field field="sales_order_id" queryOperator="="/>
    ...
    <bm:query-field field="status" queryOperator="="/>
    <bm:query-field field="closed_flag" queryOperator="="/>
  </bm:query-fields>
  <bm:relations>
    <bm:relation name="emp" joinType="LEFT OUTER" refAlias="e" refModel="fnd.FND1030.fnd_employees">
      <bm:reference foreignField="employee_id" localField="salesmen_employee_id"/>
    </bm:relation>
    <bm:relation name="partner" joinType="LEFT OUTER" refAlias="p" refModel="inv.INV2010.fnd_business_partners">
      <bm:reference foreignField="partner_id" localField="customer_id"/>
    </bm:relation>
    <bm:relation name="warehouses" joinType="LEFT OUTER" refAlias="wh" refModel="inv.INV1030.inv_warehouses">
      <bm:reference foreignField="warehouse_id" localField="issue_warehouse_id"/>
    </bm:relation>
  </bm:relations>
  <bm:ref-fields>
    <bm:ref-field name="salesmen_employee_name" relationName="emp" sourceField="employee_name"/>
    <bm:ref-field name="customer_name" relationName="partner" sourceField="partner_name"/>
    <bm:ref-field name="issue_warehouse_name" relationName="warehouses" sourceField="warehouse_name"/>
  </bm:ref-fields>
</bm:model>
```

通过这个配置文件，我们可以实现基本的数据库操作，例如根据用户输入的条件执行动态查询SQL语句，对数据进行新增、修改，或包含各种组合的批量操作，这些功能可以直接通过Web service或简单的

HTTP+JSON 方式调用。例如，在浏览器中通过 HTTP 请求：
http://localhost:8888/hap/autocrud/ord.ORD5010.ord_sales_order_headers/query?sales_order_id=45&page

即表示执行对[ord.ORD5010.ord_sales_order_headers]的查询，筛选条件为[sales_order_id=45]按[order_number]排序，分页显示，每页[10]条记录。这个查询会返回一段JSON格式的数据：

```
{"result":{"record":{"total_amount":"-19705.93","order_number":"WW2012101201","company_id":22,"customer_desc":"BW-百威","closed_flag":0}}
```

如果以SOAP的方式发起请求：

http://localhost:8888/hap/autocrud/ord.ORD5010.ord_sales_order_headers/query?sales_order_id=45&page

则会得到标准的XML格式的结果：

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <ord.ORD5010.ord_sales_order_headers success="true">
      <record total_amount="-19705.93" order_number="WW2012101201" company_id="22" customer_desc="BW-百威" closed_flag="0">
      </ord.ORD5010.ord_sales_order_headers>
    </soapenv:Body>
  </soapenv:Envelope>
```

对于 [http://localhost:8888/hap/autocrud/ord.ORD5010.ord_sales_order_headers]/insert 、
 [http://localhost:8888/hap/autocrud/ord.ORD5010.ord_sales_order_headers]/update 、
 [http://localhost:8888/hap/autocrud/ord.ORD5010.ord_sales_order_headers]/delete分别表示对
 数据执行新增、更新、删除操作；而
 [http://localhost:8888/hap/autocrud/ord.ORD5010.ord_sales_order_headers]/batch_update则表示
 对一组数据进行批量操作，可同时执行新增或更改。

2.2. 灵活的扩展机制

使用框架自动生成的SQL，虽然便利，但总会遇到一些特殊的需求，使得框架标准功能所产生的SQL无法直接使用。例如：执行删除操作时不是物理删除而是做标记；执行更改时需要同时保存历史记录；需要利用数据库的本地SQL扩展语法等。这些需求是与特定的产品、项目或使用场景相关的，框架本身无法穷举各种可能。Aurora BM提供了一种面向特性的扩展机制，使得开发者可以针对这类特殊需求开发插件式的代码。Aurora提供的基于数据耦合，事件驱动，过程抽象的扩展机制，使得这类插件代码非常简洁。例如，为利用Oracle数据库的sequence特性去生成主键，只需要在原有的BM文件中添加一句配置：<o:sequence-pk xmlns:o="...." /> 这样，生成的insert语句就由原先的标准SQL：

```
insert into table_name (f1,f2, ..., fn) values (v1, v2, ... , vn)
变为 begin insert into table_name
(f1, f2, ..., fn) values (f1_sequence.nextval, v2, ... , vn) returning f1 into :p1; end;
```

2.3. 固态的对象 vs 液态的数据

与常见的O/R映射工具不同的是，Aurora不使用静态java建模，没有关系数据库映射到java对象这样的步骤，而是以纯粹的数据容器作为数据的载体。应用开发者只要完成BM配置文件，保存到服务器约定的路径，就得到一组JSON或XML数据服务，可以立即在javascript或其他支持web service的语言中，对这些服务进行调用。对BM的修改也是即时生效，没有编译、重部署的步骤。我们甚至可以在运行时期动态生成这样的BM，或修改原有BM的配置，以实现特殊的需求——例如，通过植入一个拦截器，在BM文件被解析之前，动态删除当前用户没有权限去访问的字段，来实现可配置的列安全控制。

使用纯容器而不是对象作为数据的承载体，一个显而易见的优势就是灵活性。传统的O/R Mapping机制所产生的对象，其结构是编译时期决定的，而容器所承载的数据可以在运行时期改变其内容与结构。Aurora提供了多种数据转换的处理器，可以将来自关系数据库的单层结构，转换为多级分组结构、树形层次结构、哈希表结构，或者对数据进行行列转置。我们来看下面的例子：

```
<t:group-transform groupfield="department_id" source="/model/source_emp_list" subgroupname="enabled_flag" target="/model/t
```

这些转换是完全通过配置来实现的，并且可以自由组合叠加。相比之下，静态的Java对象构成的模型无法在运行时期改变其固有结构，不能动态地增加或删除字段，也无法将其数据结构由数组变成多级层级。如果你需要做一个按客户、订单、订单行这样用三个层次去展示数据的界面，你就必须事先建立客户包含多个订单，每个订单包含多个订单行这样的对象层级结构；如果某一天需求改变了，要在客户中间增加一层收货地址，变为4级结构，你必须再回头修改多个Java对象的结构，修改一系列映射配置文件，重新编译、部署。而在Aurora中，你需要的仅仅是为层级数据转换器增加一行配置。

2.4. 接口抽象 vs 过程抽象

作为面向对象开发方式强调接口抽象的补充，Aurora建立了一种过程抽象的机制。Aurora将数据从存储层的获取到展示层的输出（假如我们把为web service提供的XML数据也看成是一种特定类型的输出），看做是一个数据容器被一系列彼此完全独立的处理器依次处理的过程，而非传统面对对象模式下一系列对象方法调用的组合。下图展示了一个典型操作的例子。首先，前置处理器将当前用户的id、角色、登录语言等信息放入容器中，第一个处理器执行SQL查询，将数据从JDBC结果集转换为List+Map结构的复合数据容器，每条记录对应一个Map，整个数据集对应一个List；第二个处理器对数据进行多语言的转换，将数据库中按代码存储的字段，翻译成当前用户所使用的语言下实际的描述；第三个处理器进行权限控制，将当前用户没有权限查看的字段从Map中移除；第四个处理器进行数据结构的转换，将数据由一维链表转换为多级层次结构，以适应展示层界面的需要。

在这个过程中，数据由数据库中提取，到最终展示，要经过那些处理步骤，每个步骤由谁来处理，完全是通过配置而非硬编码来实现的。开发人员可以在任意一点截获控制，加入自己的处理器，实现特殊的需求。每个处理器都是功能单一、高度内聚、结构简单、易于维护的，并且具有很高的可重用性。处理器之间，通过数据容器进行参数的传递，实现了最为松散的耦合级别——数据耦合。

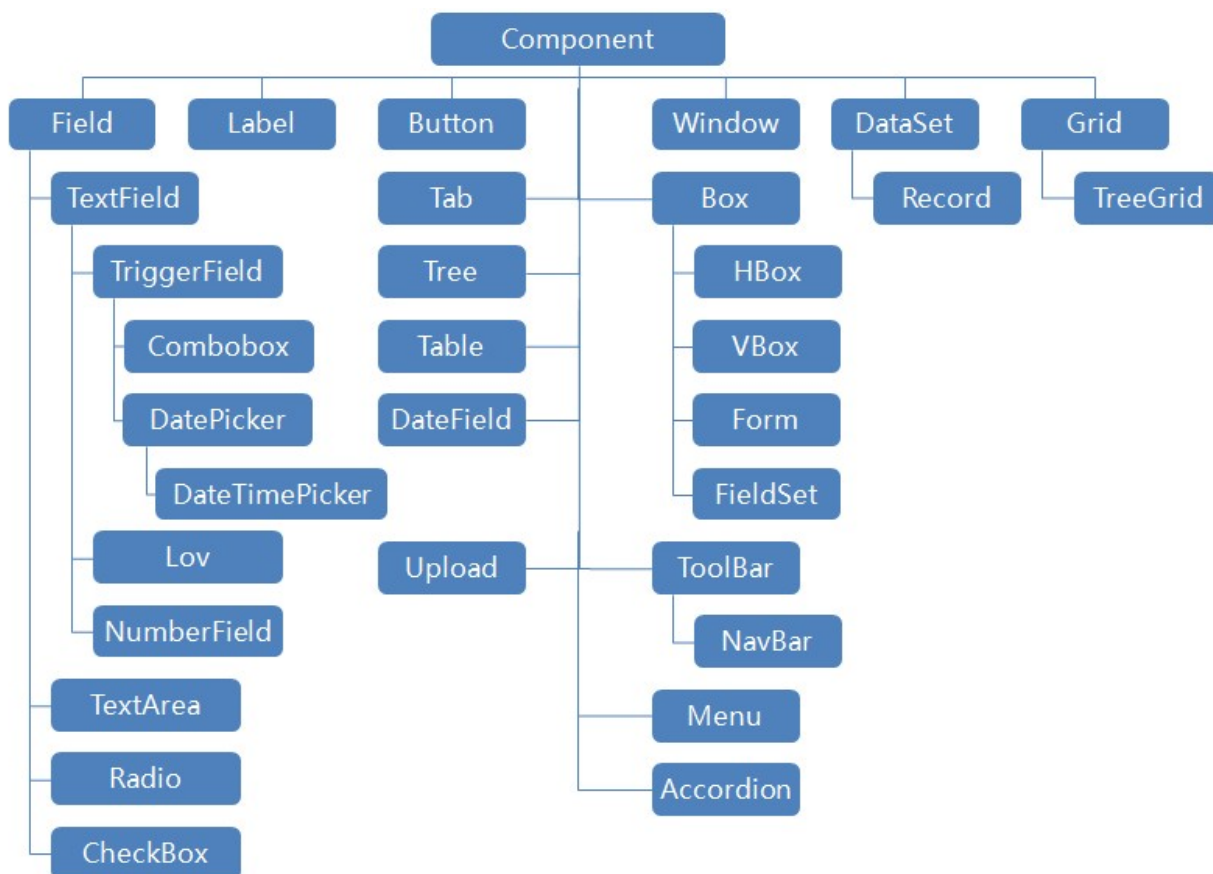
相比传统的O/R Mapping、存储层与展示层之间通过对象来传递数据、展示层调用对象的方法来存储属性这种完全依赖于接口调用的开发模式，Aurora配置驱动、面向数据、过程抽象+处理器的实现机制，体现了无与伦比的灵活性，实现了更高程度的代码复用。

第 3 章 Aurora展示层 (APL)

3.1. APL (Aurora Presentation Layer) 概述

APL提供了一整套跨浏览器的UI组件库. 所有的UI组件都是基于JavaScript和Html构建的, 运行在客户端浏览器中. 用户可以在客户端通过调用相应的函数改变UI组件的状态和行为.

APL UI组件结构图:



上图展示了所有APL UI组件的继承关系.

3.2. APL服务端工作原理

APL服务端简介

3.2.1. 基于组件重用的开发模型

Aurora展示层的主要设计场景, 是针对具有一定比例相似元素的界面。例如, 企业应用中常见的数据查询、维护功能, 它们多半都具有表单, 标签页, 表格等元素。在这种场景下, 以重用现成组件的方式来构造界面, 将比从头编码的方式更加高效。此外, 大规模重用组件至少还有以下几个好处:

- 1、在一定程度上提高代码的质量。因为可重用组件通常都是经过充分的测试, 和大量实际项目验证。
- 2、在多人开发的环境下, 更容易实现界面外观、风格的一致性。

3、让代码更容易维护。如果希望修改grid的DOM结构，只需要修改grid组件涉及的代码，而不需要修改每一处使用grid的界面。

3.2.2. 简单高效的组件模型

Aurora界面组件的工作模式，可以简单地归纳为：输入组件的运行时期配置（例如，表格的高度，宽度），以及组件绑定的数据模型（例如，一个代表员工清单的数据集），由组件经过加工，输出文本字节流（HTML/SVG/VML...）。下面的例子中，我们开发一个用于显示金额的组件currency-label，字体加粗，对于负数显示为红色，这个组件可以绑定到数据集的任意一个数字型字段。

使用这个组件很简单：

```
<a:currencyLabel name="count1" bindTarget="demo_ds" prompt="金额"/>
```

它将产生这样的输出：



为便于彻底分离组件的外观与核心逻辑，我们可以将html部分放入模板：

currency-label.tplt

```
<DIV class="${wrapClass}" style="width:${width}px;${style}" id="${id}"></DIV>
<script>
    new Aurora.Label(${config});
    ${binding}
</script>
```

CurrencyLabel.java

```
public class CurrencyLabel extends Label {

    private static final String DEFAULT_RENDERER = "Aurora.formatMoney";
    private static final String DEFAULT_TEMPLATE = "currencyLabel.tplt";

    public void onCreateViewContent(BuildSession session, ViewContext context) throws IOException {
        context.setTemplate(session.getTemplateName(DEFAULT_TEMPLATE));
        CompositeMap view = context.getView();

        String renderer = view.getString(PROPERTY_RENDERER);
        if(renderer==null)view.putString(PROPERTY_RENDERER, DEFAULT_RENDERER);
        super.onCreateViewContent(session, context);
    }
}
```

Aurora提供组件皮肤切换的支持，所有资源文件，不论是在服务端使用的模板，还是客户端引用的javascript或css，都可以在不同皮肤下有自己的版本。如果我们希望我们的currency-label看起来更酷一点，可以为它开发一个新的皮肤，为此只需要重新构造模板文件和样式表：

currency-label.css

```
.item-label-red{
height:22px;
color:red;
font-weight:bold;
}
```

currency-label.tplt

```
<DIV class="${wrapClass}" style="width:${width}px;${style}" id="${id}"></DIV>
<script>
    (function() {
        var label = new Aurora.Label(${config});
        ${binding}
        var value = label.record.get(label.binder.name);
        if(Number(value)<0) {
            label.wrap.addClass('item-label-red');
        }else{
            label.wrap.removeClass('item-label-red');
        }
    })();
</script>
```

如果一个组件包含下级组件，它将递归地调用下级组件的渲染过程，但它可以不需要了解下级组件的任何细节，因此，第三方组件可以很好地与现成组件搭配使用。下面的例子中，我们显示通过数据库查询获得的员工信息。这里使用aurora标准组件库中的repeater组件对数据集进行循环，每条记录使用box组件对字段进行布局，再使用前面我们自行开发的currency-label组件显示员工工资。

界面配置代码：

```
<a:repeater datamodel="/model/emp_list">
    <a:hBox labelWidth="100">
        <label>工资: </label>
        <a:viewItem format="###,###.00" style="font-weight:bold" value="@total_amount"/>
    </a:hBox>
</a:repeater>
```

产生的效果：

```
工资: 415448
工资: 8546.98
工资: 33401.16
工资: 24570
工资: 595.93
```

3.2.3. 无状态组件

Aurora组件的在运行时期的渲染过程，非常类似于一组静态函数调用，所有状态数据都通过参数来传递，组件内部不存在任何与session有关的状态数据。这和JSF这样重量级的界面组件模型有本质不同。JSF会在server端维持一份客户端组件的copy，这些组件的生命周期与session有关；而Aurora的组件没有复杂的生命周期，每个界面组件在一个应用内只有一个实例，在构造函数中完成一些全局性的初始化工作，之后对于任何客户端的访问，在任何界面中的使用，都是调用组件的同一个方法去渲染输出，这使得Aurora对server资源的占用显著减少。

那么，诸如“刷新后依然显示当前用户选中的菜单项”这样的需求如何实现呢？Aurora将它留给应用开发者。可以用url参数，cookie，数据库，或其它任何可能的手段，但这个特性不再是框架的一级公民。在AJAX大行其道的今天，应用系统的架构更倾向于server产生初始的html/javascript界面，提供JSON或XML形式的API，由javascript负责处理客户端的逻辑，大量的纯客户端状态数据存在于浏览器中。

因此，Aurora放弃了在server端维持界面组件状态的特性，换来的是更好的性能、更好的水平可扩展性、更简洁的组件模型。

3.2.4. 面向数据

Aurora与Java世界其它展示层组件体系的最大区别，就在于Aurora是以纯数据容器作为模型层展示层的交互媒介，而非Java对象。在Aurora组件看来，它所需要展示的数据，每条记录就是一个Map，每个字段就是Map中的一个key-value映射，记录集就是List<Map>，具有层次结构的数据就是包含List<Map>的Map（也就是我们上面例子中使用到的CompositeMap）。

使用纯数据容器作为模型层的优点：

1. 展示层与模型层彻底解耦。展示层组件需要的只是一个容器，以及如何从这个容器里取数的路径信息。至于容器里的数据是来自于关系数据库，文件，还是通过web service调用获得，完全不用关心。
2. 数据的结构可以在运行时期被修改：增加/删除字段，更改结构，获取更大的灵活性。这些灵活性对于变化频繁的展示层需求来说，是非常有用的。我们来看下面的例子，界面A以列表方式显示员工清单，每个员工一条记录：

Demol		
员工工号	员工名	部门id
EMP01	莫言	49
EMP02	李四	44
EMP05	王五	46

如果希望改变界面布局，按部门分组显示，如下图：

员工工号	员工名	部门id
EMP01	莫言	49
员工工号	员工名	部门id
EMP02	李四	44
员工工号	员工名	部门id
EMP05	王五	46

在上一章中，我们已经看到了Aurora模型层是如何能够灵活地通过配置改变数据结构，这时就用得上。我们所需做的，只是在原有的数据查询操作之后，增加一个分组转换的转换器：

```
<t:group-transform groupfield="department_id" source="/model/source_emp_list" subgroupname="enabled_flag" target="/model/
```

然后修改界面配置，在原来的列表之外增加一层repeater：

```
<a:repeater datamodel="/model/target_emp_list/a">
  <a:listView datamodel="enabled_flag">
    <a:columns>
      <a:column name="employee_code" align="center" prompt="员工工号"/>
      <a:column name="employee_name" align="center" prompt="员工名"/>
      <a:column name="department_id" align="center" prompt="部门id"/>
    </a:columns>
  </a:listView>
</a:repeater>
```

完全不需要修改模型层的java代码（因为根本就没有这样的代码），重新编译。

3.2.5. 服务端组件 + 客户端组件

在浏览器前端能力越来越被重视的今天，是否开发者只需要找到一个强大的javascript组件库，然后写html和javascript就足够了？实际上，Aurora还可以为前端开发提供一些更好的支持：

1. 按需加载资源文件。当一个组件在界面上被使用的时候，它所需要的javascript和css才会被引用到页面中。对于组件开发者来说，只需要声明自己的组件需要哪些js或css就行了；对于应用开发者来说更简单，只需要将组件标签放在界面中。
2. 根据需要在server端生成DOM结构。确实有一些纯javascript的界面组件库，它们完全在浏览器中动态生成所有需要的DOM对象，但并非所有场景都适合于这种模式。性能不够好，容易造成内存泄露，需要花费更多时间精力去保证各种版本浏览器的兼容性，这些都是难以解决的问题。Aurora组件可以根据配置，在server端动态生成所需的DOM结构，以及与之配套的javascript，组件开发者可以自由选择哪些DOM在server端生成，哪些在客户端生成。
3. 实现一些必需在server端完成的界面控制逻辑。由于aurora组件是基于配置的，而配置又是可以在运行时期动态修改的，因此开发者可以很容易地在界面渲染之前，执行一些诸如“去掉当前用户没有权限查看的表格列”这样的操作，而这类与安全性有关的操作，仅仅用javascript在客户端实现是不

够的。

3.2.6. Aurora如何创建用户界面

正文...

3.2.7. 数据容器

正文...

3.2.8. 组件映射

正文...

3.2.9. BuildSession

正文...

3.2.10. 事件机制

正文...

3.2.11. 资源文件处理

正文...

3.2.12. 动态组件配置

正文...

3.3. APL UI组件

APL 提供了各种丰富的UI组件，例如布局组件 (HBox, VBox, Form...), 数据展现组件 (Grid, Table, TreeGrid等), 数据容器组件 (DataSet等)。

APL UI组件兼容以下类型的浏览器



3.3.1. 核心Javascript API

APL采用了第三方的开源库Ext Core 3.0 (<http://www.sencha.com/products/extcore/>)作为底层的开发库,在此基础上构建了丰富的组件库。

3.3.2. DataSet

什么是DataSet?

DataSet是一个运行在客户端浏览器中的组件,本质上来说是一个JavaScript构建的对象. DataSet是一个数据容器,他封装了常用的一些数据操作,我们可以通过下面这张图来更深刻的理解DataSet的含义



通过上图我们可以看到DataSet是一个数据容器,它包含了一个数组对象用来存放所有的record对象.

Record代表一条数据对象,DataSet和Record的关系我们可以这样理解:假如我们把dataset比作数据库中的一张表,那么record就是表中的一行记录.

3.3.2.1. DataSet定义

在screen文件中我们通过<a:DataSet>标签来定义一个dataset对象

```
<a:DataSet model="sys.sys_user" id="sys_user_create_ds">
  <a:fields>
    <a:field name="user_name" required="true"/>
    <a:field name="start_date" required="true" validator="dateValidator"/>
    <a:field name="description" required="true"/>
  </a:fields>
  <events>
    <a:event name="submitsuccess" handler="onCreateUserSuccess"/>
    <a:event name="update" handler="onUpdate"/>
  </events>
</a:DataSet>
```

每一个dataset都应该定义一个id属性,在整个screen文件中不得出现重复的id值. 定义了id值我们可以在页面脚本中通过\$('sys_user_create_ds')的方式 获取到这个dataset对象,进而可以调用相应的函数方法.

fields子节点定义了这个dataset都包含哪些field以及field中的特性. 在field上我们要指定它的name,通过ajax获取到的json数据会根据name来匹配. 在field上我们还可以定义一些其他的附加特性,

例如是否必输, 是否只读等等.

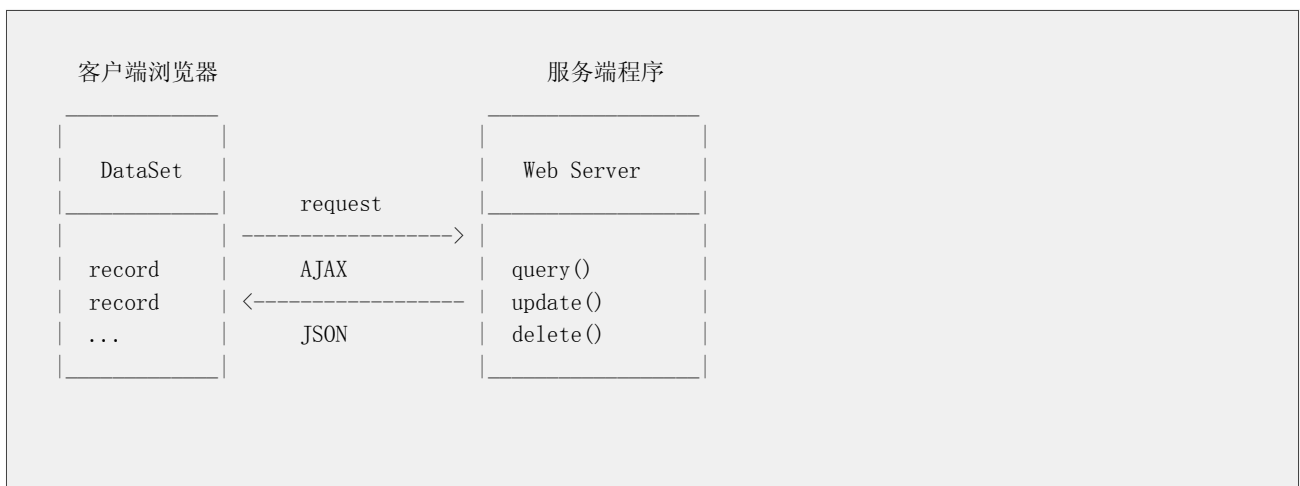
events标签定义了需要响应的事件函数, 例如update事件, 当dataset对其中一条record的field进行更新的时候, dataset会触发一个update事件, 这样我们可以通过配置一个客户端函数onUpdate来响应.

```
function onUpdate(ds, record, name, value) {
  if(name == 'user_password' || name == 'user_password_re') {
    record.validate((name == 'user_password') ? 'user_password_re' : 'user_password');
  }
  if(name == 'start_date' || name == 'end_date') {
    record.validate((name == 'start_date') ? 'end_date' : 'start_date');
  }
}
```

更多详细配置可以参考AuroraTagDocument文档

3.3.2.2. 前言

DataSet是一个客户端的JavaScript组件, 它封装了相应的ajax操作, 用来和服务端进行数据交互.



DataSet是一个客户端的JavaScript组件, 它封装了相应的ajax操作, 用来和服务端进行数据交互.

DataSet提供了基本的数据操作, 主要分为两类. 一类是客户端操作, 一类是和服务端通信.

- 客户端操作 -- 例如当调用add函数后, 其实本质上是在客户端dataset中增加一条record记录, 这个并没有同步到服务端数据库中.
- AJAX操作 -- 通过AJAX调用和服务端进行通讯, 例如query查询服务端返回相应的json数据, 填充到客户端.

3.3.2.3. 元数据 (Metadata)

Metadata元数据主要用来描述field的附加特性, 例如是否只读, 是否必输等等.

例如: 设置某一字段必输, 我们可以在配置dataset的时候指定对应field的required值

```
<a:dataset id="exp_employee_group_result_ds">
```

```

<a:fields>
  <a:field name="expense_user_group_code" required="true"/>
  <a:field name="description" required="true"/>
</a:fields>
</a:dataset>

```

例如:设置某一字段只读,我们可以在配置dataset的时候指定对应field的readOnly值

```

<a:dataset id="exp_employee_group_result_ds">
  <a:fields>
    <a:field name="expense_user_group_code" readOnly="true"/>
    <a:field name="description" required="true"/>
  </a:fields>
</a:dataset>

```

3.3.2.4. 校验 (Validate)

很多情况下我们需要对dataset的值进行校验,这个时候我们可以通过在field上配置校验函数 (validator) 来实现

例如:我们对2个日期字段进行校验,规则是结束日期不得小于开始日期. 首先我们在dataset的2个日期field上配置validator

```

<a:dataset id="fnd_companies_create_ds" model="fnd.fnd_companies">
  <a:fields>
    ...
    <a:field name="start_date_active" datatype="date" required="true" validator="dateValidator"/>
    <a:field name="end_date_active" datatype="date" validator="dateValidator"/>
  </a:fields>
</a:dataset>

```

接下来我们需要实现校验函数dateValidator

```

function dateValidator(record, name, value){
  if(name == 'start_date_active' || name == 'end_date_active'){
    var start_date = record.get('start_date_active');
    var end_date = record.get('end_date_active');
    if(typeof(end_date) != 'undefined' && !Ext.isEmpty(end_date)){
      if(!compareDate(start_date, end_date)){
        return '开始时间不能大于结束时间';
      }
    }
  }
  return true;
}

```

2个日期field公用了同一个校验函数,所以首先要判断name值,然后分别通过record获取对应的开始和结束日期. 如果校验成功返回true,校验失败返回提示信息.

3.3.2.5. 动态界面逻辑处理

待更新...

3.3.2.6. DataSet常用操作与事件

DataSet 常用函数

表 3.1. DataSet相关函数

函数	说明
add	在客户端dataset中新增一条record记录
remove	在客户端dataset中删除指定的record
query	通过指定的url查询数据, 服务端返回json数据填充到客户端dataset中
submit	降dataset中的数据提交到指定的url中

DataSet 常用事件

表 3.2. DataSet事件

事件名	说明
add	新增一条record后触发
remove	删除record后触发
update	当dataset中的record被更新后触发
load	当dataset成功加载数据后触发
submit	当dataset提交请求时触发

更多详细的函数请参考AuroraJavaScriptDocument

3.3.3. 界面布局(Layout)

APL的布局是基于服务器端生成, 这和其他的基于客户端布局的开源框架有点不同. APL的布局基本上是通过table在服务端事先生成好的, 这样的好处在于可以 减少客户端机器的压力, 充分利用服务器的资源优势.

APL的布局主要是由Box, VBox, HBox, Form, FieldSet等容器组件组成.

3.3.3.1. Box

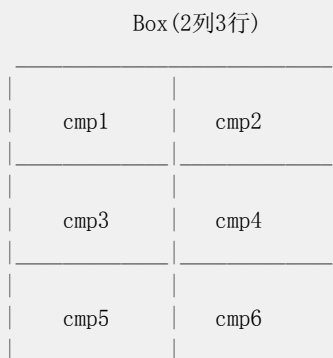
Box组件类似html中的table标签, 通过行(row)和列(column)的配置来构建一个表格.

在screen文件中我们通过<a:box>标签来定义一个box对象, 然后定义row和column.

box标签下的组件将会按照从左到右从上到下的原则进行排列。

```
<a:box column="2" row="3">
  <a:textField name="cmp1"/>
  <a:textField name="cmp2"/>
  <a:textField name="cmp3"/>
  <a:textField name="cmp4"/>
  <a:textField name="cmp5"/>
  <a:textField name="cmp6"/>
</a:box>
```

布局实例图：



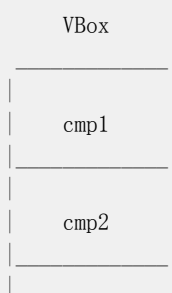
3.3.3.2. VBox

VBox组件继承自Box组件, VBox的column恒等于1, 所以VBox相当于column=1的Box组件.

VBox下的组件按照自上而下的方式进行布局

```
<a:vBox>
  <a:textField name="cmp1"/>
  <a:textField name="cmp2"/>
  <a:textField name="cmp3"/>
</a:vBox>
```

布局实例图：



cmp3

3.3.3.3. HBox

HBox组件继承自Box组件, HBox的row恒等于1, 所以HBox相当于row=1的Box组件.

HBox下的组件按照从左到右的方式进行布局

```
<a:hBox>
  <a:textField name="cmp1"/>
  <a:textField name="cmp2"/>
  <a:textField name="cmp3"/>
</a:hBox>
```

布局实例图:

HBox		
cmp1	cmp2	cmp3

3.3.3.4. Form

Form组件继承自Box组件, 带有一个title头.

注意:Form和html中的form标签是有本质区别的. Form组件仅仅是一个布局容器, 并没有提交数据的功能.

Form组件的布局方式和Box类似, 根据row和column的布局所有组件, 采取从上到下从左到右的方式.

```
<a:form id="loginForm" labelWidth="100" row="1" title="Form Title" width="320">
  <a:textField name="cmp1"/>
  <a:textField name="cmp2"/>
  <a:textField name="cmp2"/>
</a:form>
```

布局实例图:

Form		
Form Title		
cmp1	cmp2	cmp3

提示:Form和Box以及其他容器组件类似,都是可以嵌套的.举例来说Form下可以再次嵌套VBox, HBox, Form标签.

3.3.3.5. FieldSet

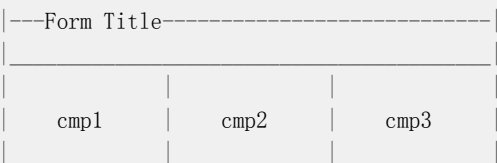
FieldSet组件继承自Box组件, 带有一个title头. 它和Form标签的不同仅仅局限在界面展现上.

提示:FieldSet的配置和Form基本类似,它和Form标签的不同仅仅局限在界面展现上.

FieldSet组件的布局方式和Box类似, 根据row和column的布局所有组件, 采取从上到下从左到右的方式.

```
<a:fieldSet id="loginForm" labelWidth="100" row="1" title="Form Title" width="320">
  <a:textField name="cmp1"/>
  <a:textField name="cmp2"/>
  <a:textField name="cmp2"/>
</a:fieldSet>
```

布局实例图：



提示:FieldSet和Box以及其他容器组件类似,都是可以嵌套的.举例来说FieldSet下可以再次嵌套VBox, HBox, Form标签.

3.3.4. 编辑组件

编辑组件是提供文本输入编辑及选择功能的组件，主要是由 TextField, NumberField, ComboBox, DatePicker, Lov, DateTimePicker, TextArea, Radio, CheckBox 等组件组成。

编辑组件也可作为Grid组件的Editor，具体请参阅Grid的编辑器。

3.3.4.1. TextField

TextField是一个提供文本输入编辑的组件，可限制大小写的输入。



上图是TextField组件在页面中的呈现，输入框前的文字信息是通过TextField标签属性prompt来定义的。

3.3.4.1.1. TextField定义

在screen文件中我们通过<a:textField>标签来定义一个TextField对象。

```
<a:textField bindTarget="login_dataset" id="user_name_tf" name="user_name"
  prompt="HAP_USERNAME" width="150" typeCase="upper">
  <a:events>
    <a:event handler="login" name="enterdown"/>
  </a:events>
</a:textField>
```

textField标签可以设置一个id属性，id是组件的唯一标识，我们可以在页面脚本中用\$('id')的方法获得该id对应的组件对象，进而可以调用相应的函数方法。

textField标签的bindTarget属性可指定一个dataset对象的id，name属性可指定该dataset其中一个field的名字。这两个属性必须联合使用，其功能是将TextField对象绑定到dataset中的一个field上，进而我们只要对dataset进行操作就能即时反映在TextField上，另外TextField上的文字编辑也会立刻修改dataset中的数据。

events标签定义了需要响应的事件函数，例如enterdown事件，当键盘键入回车键时，TextField会触发enterdown事件，这样我们可以通过配置一个客户端函数login来响应。

```
function login() {
  var lds = $('login_dataset');
  var record = lds.getCurrentRecord();
  Aurora.request({url:'login.svc', para:record.data, success:function() {
    window.location.href='role_select.screen'
  },scope:this});
}
```

3.3.4.1.2. TextField标签属性

表 3.3.

属性名	用途	默认值	是否必填
bindTarget	组件所绑定的dataset数据集，属性值是dataset的ID。		
className	组件的样式表。		
emptyText	当组件没有值的时候显示在组件上的提示信息。		
id	组件的唯一标识，可用\$(id)方法获得组件对象。		
marginWidth	组件与窗口之间的宽度差，单位像素(px)，可以根据窗口宽度的改变而改变。		
name	组件对应dataset数据集中的一个字段field，属性值是字段名。		
prompt	输入框前的提示文字，默认调用BM的prompt。		
readOnly	设定组件是否只读。 取值 true false	false	

属性名	用途	默认值	是否必填
required	设定组件是否必填。 取值 true false	false	
style	组件的样式。		
typeCase	组件的大小写输入限制。 取值 upper lower		
width	组件的宽度，单位像素(px)。	150	

3.3.4.1.3. TextField对象事件

表 3.4.

事件名	用途
blur	失去焦点时触发的事件。
change	文本内容发生改变时触发的事件。
enterdown	敲击回车键时触发的事件。
focus	获得焦点时触发的事件。
keydown	键盘按下时触发的事件。
keypress	键盘敲击时触发的事件。
keyup	键盘抬起时触发的事件。
mouseover	鼠标移到组件上时触发的事件。
mouseout	鼠标移出组件时触发的事件。

3.3.4.2. NumberField

NumberField是一个提供数字输入编辑的组件，继承自TextField组件，拥有TextField标签的属性以及TextField对象的方法和事件。

税率:

上图是NumberField组件在页面中的呈现，输入框前的文字信息是通过NumberField标签属性prompt来定义的。

3.3.4.2.1. NumberField定义

在screen文件中我们通过<a:numberField>标签来定义一个NumberField对象。

```
<a:numberField bindTarget="fnd_tax_type_codes_query_ds" name="tax_type_rate"
  allowDecimals="true" allowFormat="true" decimalPrecision="1">
  <a:events>
```

```
<a:event name="enterdown" handler="queryTaxTypeCodes"/>
</a:events>
</a:numberField>
```

numberField标签可以设置一个id属性，id是组件的唯一标识，我们可以在页面脚本中用\$('id')的方法获得该id对应的组件对象，进而可以调用相应的函数方法。

numberField标签的bindTarget属性可指定一个dataset对象的id，name属性可指定该dataset其中一个field的名字。这两个属性必须联合使用，其功能是将NumberField对象绑定到dataset中的一个field上，进而我们只要对dataset进行操作就能即时反映在NumberField上，另外NumberField上的文字编辑也会立刻修改dataset中的数据。

events标签定义了需要响应的事件函数，例如enterdown事件，当键盘键入回车键时，NumberField会触发enterdown事件，这样我们可以通过配置一个客户端函数queryTaxTypeCodes来响应。

```
function queryTaxTypeCodes() {
    $('fnd_tax_type_codes_result_ds').query();
}
```

3.3.4.2.2. NumberField标签属性

表 3.5.

属性名	用途	默认值	是否必填
allowDecimals	是否允许NumberField的值为小数。 取值 true false	true	
allowNegative	是否允许NumberField的值为负数。 取值 true false	true	
allowFormat	是否允许NumberField的值按照千分位显示。 取值 true false	true	
decimalPrecision	组件的小数位精度，必须在allowDecimals为true的情况下才能使用。	2	
其他	参阅TextField标签属性。		

3.3.4.2.3. NumberField对象事件

请参阅TextField的对象事件。

3.3.4.3. ComboBox

ComboBox是一个可输入的下拉框组件，继承自TextField组件，拥有TextField标签的属性以及TextField对象的方法和事件。其主要的功能就是运用键值对的形式将数据(键)以值的形式来呈现。



上图是ComboBox组件在页面中的呈现，输入框前的文字信息是通过ComboBox标签属性prompt来定义的。

3.3.4.3.1. ComboBox定义

在screen文件中我们通过<a:comboBox>标签来定义一个ComboBox对象。

```
<a:dataset id="sys_user_islocked_ds">
  <a:datas>
    <a:record name="已冻结" code="Y"/>
    <a:record name="未冻结" code="N"/>
  </a:datas>
</a:dataset>
<a:dataset id="sys_user_query_ds">
  <a:fields>
    <a:field name="user_name"/>
    <a:field name="frozen_flag_display" displayField="name" options="sys_user_islocked_ds"
      returnField="frozen_flag" valueField="code"/>
  </a:fields>
</a:dataset>
<a:comboBox name="frozen_flag_display" bindTarget="sys_user_query_ds" prompt="SYS_USER. IS_FROZEN">
  <a:events>
    <a:event handler="login" name="enterdown"/>
  </a:events>
</a:comboBox>
```

comboBox标签可以设置一个id属性，id是组件的唯一标识，我们可以在页面脚本中用\$('id')的方法获得该id对应的组件对象，进而可以调用相应的函数方法。

comboBox标签的bindTarget属性可指定一个dataset对象的id，name属性可指定该dataset其中一个field的名字。这两个属性必须联合使用，其功能是将ComboBox对象绑定到dataset中的一个field上，进而我们只要对dataset进行操作就能即时反映在ComboBox上，另外ComboBox上的文字编辑也会立刻修改dataset中的数据。

comboBox所绑定的dataset应该至少要有两条字段(field)，一条是对应数据库的数据的隐式字段，另一条是用来显示文字的显示字段。如上代码所示，comboBox的选项是在显示字段(如上代码中name="frozen_flag_display"的field)标签上加上returnField, options, displayField, valueField属性组合实现的。其中returnField属性是隐式字段名，表示comboBox选中选项后用哪个字段来承载这个值，进而通过dataset的方法用来对数据库进行添加修改等操作。options属性是一个选项的数据集的ID，该数据集的所有记录(record)都应该有两条字段(field)，一条字段是选项的值(如上代码中code="Y"的code)，另一条字段是选项显示的文本(如上代码中name="已冻结"的name)，valueField和displayField属性分别指定的就是这两条字段的名称(code和name)。

events标签定义了需要响应的事件函数，例如enterdown事件，当键盘键入回车键时，ComboBox会触发enterdown事件，这样我们可以通过配置一个客户端函数login来响应。

```
function login() {
  var lds = $('login_dataset');
  var record = lds.getCurrentRecord();
```

```
Aurora.request({url:'login.svc', para:record.data, success:function() {
    window.location.href='role_select.screen'
  },scope:this});
}
```

3.3.4.3.2. ComboBox标签属性

请参阅TextField的标签属性。

3.3.4.3.3. ComboBox对象事件

表 3.6.

事件名	用途
select	选择选项时触发的事件。
其他	请参阅TextField对象的事件

3.3.4.4. DateField

DateField正文 ...

3.3.4.5. DatePicker

DatePicker是一个提供日期输入编辑的组件，继承自TextField组件，拥有TextField标签的属性以及TextField对象的方法和事件。



上图是DatePicker组件在页面中的呈现，输入框前的文字信息是通过DatePicker标签属性prompt来定义的。

3.3.4.5.1. DatePicker定义

在screen文件中我们通过<a:datePicker>标签来定义一个DatePicker对象。

```
<a:datePicker name="start_date" bindTarget="sys_user_create_ds" viewSize="2"
    enableBesideDays="both" enableMonthBtn="both">
  <a:events>
    <a:event handler="login" name="enterdown"/>
  </a:events>
</a:datePicker>
```

datePicker标签可以设置一个id属性，id是组件的唯一标识，我们可以在页面脚本中用\$('id')的方法获得该id对应的组件对象，进而可以调用相应的函数方法。

datePicker标签的bindTarget属性可指定一个dataset对象的id，name属性可指定该dataset其中一个field的名字。这两个属性必须联合使用，其功能是将DatePicker对象绑定到dataset中的一个field上，进而我们只要对dataset进行操作就能即时反映在DatePicker上，另外DatePicker上的文字编辑也会立刻修改dataset中的数据。

events标签定义了需要响应的事件函数，例如enterdown事件，当键盘键入回车键时，DatePicker会触发enterdown事件，这样我们可以通过配置一个客户端函数login来响应。

```
function login() {
    var lds = $('login_dataset');
    var record = lds.getCurrentRecord();
    Aurora.request({url:'login.svc', para:record.data, success:function() {
        window.location.href='role_select.screen'
    }, scope:this});
}
```

3.3.4.5.2. 日期渲染

渲染函数(dayRenderer)

很多情况下我们需要将显示的日期表进行一些特殊处理，例如我们需要将今天之前的日期设为不能选择状态。这个时候我们就需要在datepicker标签上指定dayRenderer渲染函数来实现。

首先我们需要实现一个dayRenderer函数，来判断当前日期，然后返回一段html代码。

```
function rendererDay(cell, date, text, currentMonth) {
    var today=new Date();
    if(date < new Date(today.getFullYear(),today.getMonth(),today.getDate())){
        cell.disabled=true;
    }
    return text;
}
```

接下来在datepicker标签上指定dayRenderer属性为rendererDay.

```
<a:datepicker dayRenderer="rendererDay"/>
```

3.3.4.5.3. DatePicker标签属性

表 3.7.

属性名	用途	默认值	是否必填
dayRenderer	日期渲染函数。dayRenderer属性指定一个回调函数的函数名，该函数可带三个参数，依次为		

属性名	用途	默认值	是否必填
	<p>cell, date, text。</p> <p>cell - 显示日期的单元格，当 cell.disabled=true时，该日期无法被选择。</p> <p>date - 日期对应的date对象。</p> <p>text - 日期所显示的文本，函数的返回值需为包含此参数的HTML字符串。</p>		
enableBesideDays	<p>enableBesideDays属性指定当月日期表是否显示上月结尾和(或)下月开头的日期。</p> <p>取值 both none pre next</p>	both	
enableMonthBtn	<p>enableMonthBtn属性指定日期表是否显示上月按钮和(或)下月按钮。</p> <p>取值 both none pre next</p>	both	
viewSize	显示日期表的个数，已当前月开始依次排列。最大值是4。	1	

3.3.4.5.4. DatePicker对象事件

表 3.8.

事件名	用途
select	选择日期时触发的事件。
其他	请参阅TextField对象的事件

3.3.4.6. Lov

Lov是一个提供文本输入编辑和通过弹出窗口提供选项选择的组件，继承自TextField组件，拥有TextField标签的属性以及TextField对象的方法和事件。弹出窗口中为一个固定格式的页面，包含选项集的查询条件输入框和用Grid组件成列的选项集。



上图是Lov组件在页面中的呈现，输入框前的文字信息是通过Lov标签属性prompt来定义的。

3.3.4.6.1. Lov定义

在screen文件中我们通过<a:lov>标签来定义一个Lov对象。

```
<a:DataSet id="gld_exchange_rate_ds" autocreate="true">
  <a:datas dataSource="/model/gerc"/>
  <a:fields>
    <a:field name="currency_code_frn" lovGridHeight="300" lovHeight="460"
      lovService="gld.gld_currency_lov?currency_code_frn=${/model/gerc/record/@currency_code}"
      lovWidth="490" title="币种选择">
      <mapping>
        <map from="currency_code" to="currency_code_frn"/>
        <map from="currency_name" to="currency_name_frn"/>
      </mapping>
    </a:field>
  </a:fields>
  <a:field name="currency_name_frn" readonly="true"/>
</a:DataSet>

<a:lov name="currency_code_frn" bindTarget="gld_exchange_rate_ds" prompt="GLD_CURRENCY.CURRENCY_CODE_FRN">
  <a:events>
    <a:event handler="login" name="enterdown"/>
  </a:events>
</a:lov>
```

lov标签可以设置一个id属性，id是组件的唯一标识，我们可以在页面脚本中用\$('id')的方法获得该id对应的组件对象，进而可以调用相应的函数方法。

lov标签的bindTarget属性可指定一个dataset对象的id，name属性可指定该dataset其中一个field的名字。这两个属性必须联合使用，其功能是将Lov对象绑定到dataset中的一个field上，进而我们只要对dataset进行操作就能即时反映在Lov上，另外Lov上的文字编辑也会立刻修改dataset中的数据。

如上代码，field标签是lov所绑定的dataset中的一条字段(field)，field标签中lovHeight, lovWidth属性可指定弹出窗口的高度和宽度，lovService属性可指定弹出窗口中生成页面的BM，title属性可指定弹出框的标题。注意：以上属性均可定义在lov标签上，效果相同，但是建议定义在field标签上。

field标签下的mapping标签定义了弹出窗口选项集的字段和主窗口的字段的联系关系，每条map标签都有from和to属性，from是lov弹出窗口中选项的字段，to是lov在主窗口绑定的dataset中的字段。

events标签定义了需要响应的事件函数，例如enterdown事件，当键盘键入回车键时，Lov会触发enterdown事件，这样我们可以通过配置一个客户端函数login来响应。

```
function login() {
  var lds = $('login_dataset');
  var record = lds.getCurrentRecord();
  Aurora.request({url:'login.svc', para:record.data, success:function() {
    window.location.href='role_select.screen'
  },scope:this});
}
```

3.3.4.6.2. Lov标签属性

表 3.9.

属性名	用途	默认值	是否必填
lovGridHeight	弹出窗口中Grid的高度。	350	
lovHeight	弹出窗口的高度。	400	
lovService	弹出窗口中数据对应的BM。		
lovUrl	弹出窗口中页面的Url。		
lovWidth	弹出窗口的宽度。	400	
title	弹出窗口的标题。		
fetchRemote	手工输入后是否自动查询数据。		
其他	请参阅TextField的标签属性。		

3.3.4.6.3. Lov对象事件

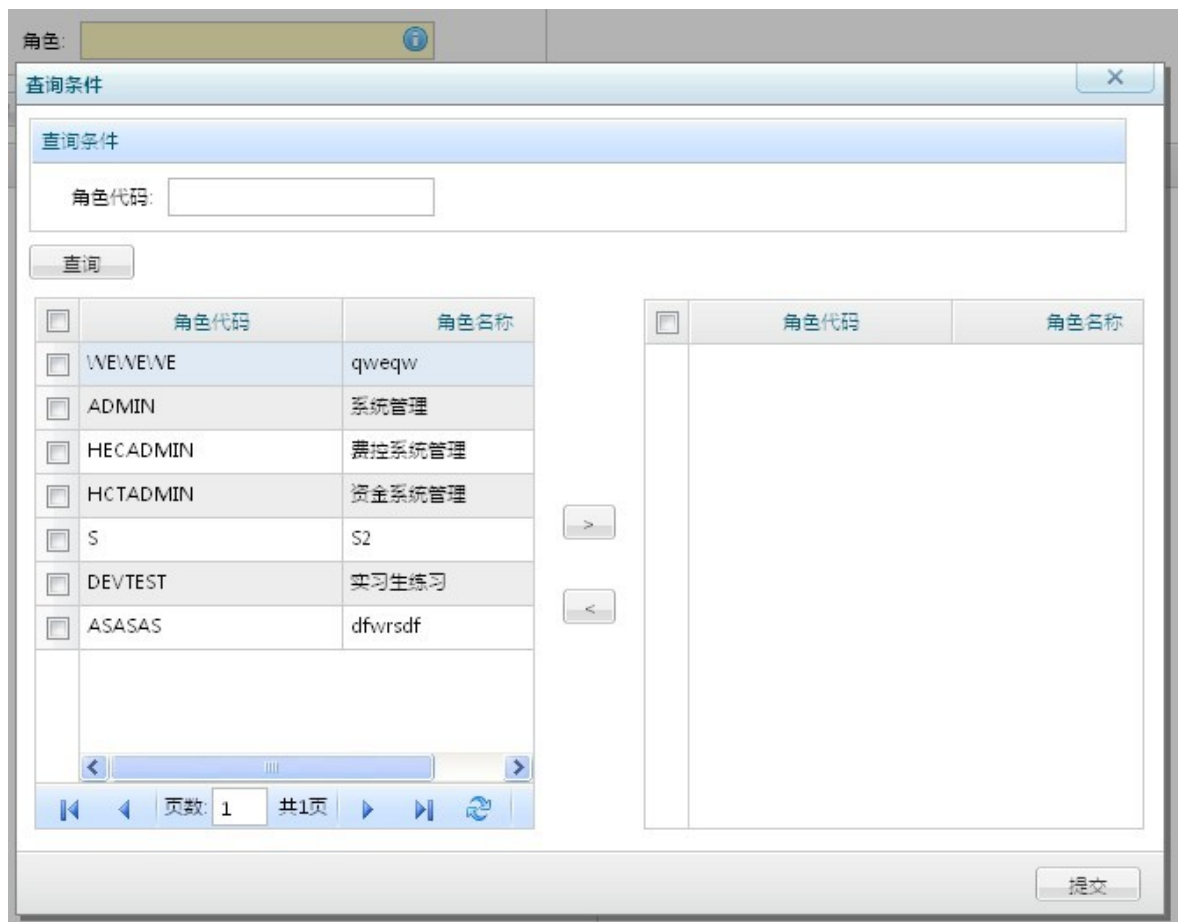
表 3.10.

事件名	用途
commit	窗口的选项被选择后触发的事件，手工输入后自动查询数据(fetchRemote=true)的时候也会触发。

事件名	用途
其他	请参阅TextField的对象事件

3.3.4.7. MultiLov

MultiLov是一个提供文本输入编辑和通过弹出窗口提供选项选择的组件，继承自Lov组件，拥有Lov标签的属性以及Lov对象的方法和事件。弹出窗口中为一个固定格式的页面，包含选项集的查询条件输入框和两个用Grid组件成列的选项集和待选集。



上图是MultiLov组件在页面中的呈现，输入框前的文字信息是通过Lov标签属性prompt来定义的。

3.3.4.7.1. MultiLov定义

在screen文件中我们通过<a:multiLov>标签来定义一个MultiLov对象。

```
<a:DataSet id="gld_exchange_rate_ds" autocreate="true">
  <a:datas dataSource="/model/gerc"/>
  <a:fields>
    <a:field displayField="currency_name" valueField="currency_code" name="currency_code_frn"
      lovGridHeight="300" lovHeight="460" lovWidth="490" title="币种选择"
      lovService="gld.gld_currency_lov?currency_code_frn=${/model/gerc/record/@currency_code}"/>
  </a:fields>
  <a:field name="currency_name_frn" readonly="true"/>
</a:DataSet>

<a:multiLov name="currency_code_frn" bindTarget="gld_exchange_rate_ds"
  prompt="GLD_CURRENCY.CURRENCY_CODE_FRN">
```

```

<a:events>
  <a:event handler="login" name="enterdown"/>
</a:events>
</a:multiLov>

```

multiLov标签可以设置一个id属性，id是组件的唯一标识，我们可以在页面脚本中用\$('id')的方法获得该id对应的组件对象，进而可以调用相应的函数方法。

multiLov标签的bindTarget属性可指定一个dataset对象的id，name属性可指定该dataset其中一个field的名字。这两个属性必须联合使用，其功能是将Lov对象绑定到dataset中的一个field上，进而我们只要对dataset进行操作就能即时反映在MultiLov上，另外MultiLov上的文字编辑也会立刻修改dataset中的数据。

如上代码，field标签是multiLov所绑定的dataset中的一条字段(field)，field标签中lovHeight, lovWidth属性可指定弹出窗口的高度和宽度，lovService属性可指定弹出窗口中生成页面的BM，title属性可指定弹出框的标题。注意：以上属性均可定义在multiLov标签上，效果相同，但是建议定义在field标签上。

field标签中displayField和valueField属性定义了弹出窗口选项集的字段和主窗口的字段的联系关系，displayField是将multiLov弹出窗口中选项的对应字段的值显示在主窗口中的输入框中，valueField是将弹出窗口中选项的对应字段的值传递给主窗口的控件，用于数据传递。由于传回的多个数据，显示的格式是以分号分隔，而值是以单引号括起以及逗号分隔

events标签定义了需要响应的事件函数，例如enterdown事件，当键盘键入回车键时，MultiLov会触发enterdown事件，这样我们可以通过配置一个客户端函数login来响应。

```

function login(){
  var lds = $('login_dataset');
  var record = lds.getCurrentRecord();
  Aurora.request({url:'login.svc', para:record.data, success:function(){
    window.location.href='role_select.screen'
  },scope:this});
}

```

3.3.4.7.2. MultiLov标签属性

表 3.11.

属性名	用途	默认值	是否必填
displayField	MultiLov弹出框中返回到输入框上用来显示的字段		
valueField	MultiLov弹出框中返回到输入框上用来表示控件值的字段		
其他	请参阅Lov的标签属性。		

3.3.4.7.3. MultiLov对象事件

表 3.12.

事件名	用途
commit	弹出窗口中选择的选项集提交后触发的事件。
其他	请参阅Lov的对象事件

3.3.4.8. DateTimePicker

DateTimePicker是一个提供日期及时间输入编辑的组件，继承自DatePicker组件，拥有DatePicker标签的属性以及DatePicker对象的方法和事件。



上图是DateTimePicker组件在页面中的呈现，输入框前的文字信息是通过DateTimePicker标签属性prompt来定义的。

3.3.4.8.1. DateTimePicker定义

在screen文件中我们通过<a:dateTimePicker>标签来定义一个DateTimePicker对象。

```
<a:dateTimePicker name="start_date" bindTarget="sys_user_create_ds" viewSize="2"
  enableBesideDays="both" enableMonthBtn="both">
  <a:events>
    <a:event handler="login" name="enterdown"/>
  </a:events>
</a:dateTimePicker>
```

dateTimePicker标签可以设置一个id属性，id是组件的唯一标识，我们可以在页面脚本中用\$('id')的方法获得该id对应的组件对象，进而可以调用相应的函数方法。

dateTimePicker标签的bindTarget属性可指定一个dataset对象的id，name属性可指定该dataset其中一个field的名字。这两个属性必须联合使用，其功能是将DateTimePicker对象绑定到dataset中的一个field上，进而我们只要对dataset进行操作就能即时反映在DateTimePicker上，另外DateTimePicker上的文字编辑也会立刻修改dataset中的数据。

events标签定义了需要响应的事件函数，例如enterdown事件，当键盘键入回车键时，DateTimePicker会触发enterdown事件，这样我们可以通过配置一个客户端函数login来响应。

```
function login() {
  var lds = $('login_dataset');
```

```
var record = lds.getCurrentRecord();
Aurora.request({url:'login.svc', para:record.data, success:function() {
    window.location.href='role_select.screen'
}},scope:this});
}
```

3.3.4.8.2. 日期渲染

渲染函数(dayRenderer)

很多情况下我们需要将显示的日期表进行一些特殊处理，例如我们需要将今天之前的日期设为不能选择状态。这个时候我们就需要在datetimepicker标签上指定dayRenderer渲染函数来实现。

首先我们需要实现一个dayRenderer函数，来判断当前日期，然后返回一段html代码。

```
function renderDay(cell, date, text, currentMonth) {
    var today=new Date();
    if(date < new Date(today.getFullYear(), today.getMonth(), today.getDate())){
        cell.disabled=true;
    }
    return text;
}
```

接下来在datetimepicker标签上指定dayRenderer属性为renderDay。

```
<a:dateTimePicker dayRenderer="renderDay"/>
```

3.3.4.8.3. DateTimePicker标签属性

请参阅DatePicker的标签属性。

3.3.4.8.4. DateTimePicker对象事件

请参阅DatePicker的对象事件。

3.3.4.9. TextArea

TextArea是一个提供多行文本输入编辑的组件。

SQL验证:



上图是TextArea组件在页面中的呈现，输入框前的文字信息是通过TextArea标签属性prompt来定义的。

3.3.4.9.1. TextArea定义

在screen文件中我们通过<a:textArea>标签来定义一个TextArea对象。



```
<a:textArea name="sql_validation" id="sql_v" bindTarget="sys_parameter_define_ds" width="350">
  <a:events>
    <a:event handler="login" name="enterdown"/>
  </a:events>
</a:textArea>
```

textArea标签可以设置一个id属性，id是组件的唯一标识，我们可以在页面脚本中用\$('id')的方法获得该id对应的组件对象，进而可以调用相应的函数方法。

textArea标签的bindTarget属性可指定一个dataset对象的id，name属性可指定该dataset其中一个field的名字。这两个属性必须联合使用，其功能是将TextArea对象绑定到dataset中的一个field上，进而我们只要对dataset进行操作就能即时反映在TextArea上，另外TextArea上的文字编辑也会立刻修改dataset中的数据。

events标签定义了需要响应的事件函数，例如enterdown事件，当键盘键入回车键时，TextArea会触发enterdown事件，这样我们可以通过配置一个客户端函数login来响应。

```
function login() {
  var lds = $('login_dataset');
  var record = lds.getCurrentRecord();
  Aurora.request({url:'login.svc', para:record.data, success:function() {
    window.location.href='role_select.screen'
  }, scope:this});
}
```

3.3.4.9.2. TextArea标签属性

可参阅TextField的标签属性。

3.3.4.9.3. TextArea对象事件

可参阅TextField的对象事件。

3.3.4.10. Radio

Radio是一组单项选择按钮组件。



上图是Radio组件在页面中的呈现。

3.3.4.10.1. Radio定义

在screen文件中我们通过<a:radio>标签来定义一个Radio对象。

```
<a:radio name="state" bindTarget="sys_user_create_ds" layout="vertical"
style="padding-top:5px;padding-bottom:5px;" width="80">
  <a:items>
    <a:item label="SYS_USER.PASSWD_EXPIRED_DAYS" value="1"/>
    <a:item label="SYS_USER.PASSWD_EXPIRED_TIMES" value="2"/>
  </a:items>
</a:radio>
```

```

    <a:item label="SYS_USER.PASSWD_EXPIRED_NEVER" value="3"/>
  </a:items>
  <a:events>
    <a:event name="change" handler="onRadioChange"/>
  </a:events>
</a:radio>

```

radio标签可以设置一个id属性，id是组件的唯一标识，我们可以在页面脚本中用\$('id')的方法获得该id对应的组件对象，进而可以调用相应的函数方法。

radio标签的bindTarget属性可指定一个dataset对象的id，name属性可指定该dataset其中一个field的名字。这两个属性必须联合使用，其功能是将Radio对象绑定到dataset中的一个field上，进而我们只要对dataset进行操作就能即时反映在Radio上，另外Radio上的文字编辑也会立刻修改dataset中的数据。

items标签定义了radio的选项组。items标签下每个item标签即一个选项，label属性指定选项后面的提示性息，value属性指定了选项的值。

events标签定义了需要响应的事件函数，例如change事件，当选中的选项改变为选中另一个选项时，Radio会触发change事件，这样我们可以通过配置一个客户端函数onRadioChange来响应。

```

function onRadioChange(radio, newValue, oldValue){
  var record = $('sys_user_create_ds').getCurrentRecord();
  if(newValue=='1'){
    record.set('password_lifespan_access', null)
    record.getMeta().getField('password_lifespan_days').setReadOnly(false);
    record.getMeta().getField('password_lifespan_access').setReadOnly(true);
  }else if(newValue=='2'){
    record.set('password_lifespan_days', null)
    record.getMeta().getField('password_lifespan_days').setReadOnly(true);
    record.getMeta().getField('password_lifespan_access').setReadOnly(false);
  }else{
    record.set('password_lifespan_access', null)
    record.set('password_lifespan_days', null)
    record.getMeta().getField('password_lifespan_days').setReadOnly(true);
    record.getMeta().getField('password_lifespan_access').setReadOnly(true);
  }
}

```

另外，radio标签还有另一种方式来定义Radio对象，Radio的选项不是用items标签来硬性定义，而是可以用类似于combobox的方法，将选项用options属性绑定到一个dataset数据集，相应的文本提示信息和选项值用labelField和valueField来指定。

```

<a:radio id="roleRadios" labelExpression="$ {@role_description} _$ {@company_short_name}"
layout="vertical" options="/model/role_list" valueField="role_company" width="230">
  <a:events>
    <event name="enterdown" handler="goToMain"/>
  </a:events>
</a:radio>

```

labelExpression属性指定如何选项文本提示信息的表达式，可替代labelField属性。

3.3.4.10.2. Radio标签属性

表 3.13.

属性名	用途	默认值	是否必填
labelExpression	指定如何选项文本提示信息的表达式。		
labelField	指定options绑定的选项数据集中用来显示选项文本提示信息的field。	label	
layout	指定按钮组的排列方式。 取值 horizontal vertical	horizontal	
options	指定Radio选项的数据集。		
valueField	指定options绑定的选项数据集中用来表示选项值的field。		

3.3.4.10.3. Radio对象事件

表 3.14.

事件名	用途
blur	失去焦点时触发的事件。
change	选中的选项改变为选中另一个选项时触发的事件。
click	点击选项按钮时触发的事件。
enterdown	敲击回车键时触发的事件。
focus	获得焦点时触发的事件。
keydown	键盘按下时触发的事件。
mouseover	鼠标移到组件上时触发的事件。
mouseout	鼠标移出组件时触发的事件。

3.3.4.11. CheckBox

CheckBox是一个多项选择按钮组件。

启用: ☒

上图是CheckBox组件在页面中的呈现，选择框前的文字信息是通过ComboBox标签属性prompt来定义的。

3.3.4.11.1. CheckBox定义

在screen文件中我们通过<a:checkbox>标签来定义一个CheckBox对象。

```
<a:dataset id="sys_notify_edit_ds">
  <a:fields>
    <a:field name="enabled_flag" checkedValue="Y" defaultValue="Y" uncheckedValue="N"/>
  </a:fields>
</a:dataset>
<a:checkbox name="enabled_flag" bindTarget="sys_notify_edit_ds" prompt="FND_OPERATION_UNITS.ENABLED_FLAG">
  <a:events>
    <a:event name="change" handler="onChange"/>
  </a:events>
</a:checkbox>
```

checkbox标签可以设置一个id属性，id是组件的唯一标识，我们可以在页面脚本中用\$('id')的方法获得该id对应的组件对象，进而可以调用相应的函数方法。

checkbox标签的bindTarget属性可指定一个dataset对象的id，name属性可指定该dataset其中一个field的名字。这两个属性必须联合使用，其功能是将CheckBox对象绑定到dataset中的一个field上，进而我们只要对dataset进行操作就能即时反映在CheckBox上，另外改变CheckBox的选中状态也会立刻修改dataset中的数据。

events标签定义了需要响应的事件函数，例如change事件，当选中的选项改变为选中另一个选项时，CheckBox会触发change事件，这样我们可以通过配置一个客户端函数onChange来响应。

```
function onChange(checkbox, newValue, oldValue){
  var record = $('sys_user_create_ds').getCurrentRecord();
  if(newValue=='1'){
    record.set('password_lifespan_access', null)
    record.getMeta().getField('password_lifespan_days').setReadOnly(false);
    record.getMeta().getField('password_lifespan_access').setReadOnly(true);
  }else if(newValue=='2'){
    record.set('password_lifespan_days', null)
    record.getMeta().getField('password_lifespan_days').setReadOnly(true);
    record.getMeta().getField('password_lifespan_access').setReadOnly(false);
  }else{
    record.set('password_lifespan_access', null)
    record.set('password_lifespan_days', null)
    record.getMeta().getField('password_lifespan_days').setReadOnly(true);
    record.getMeta().getField('password_lifespan_access').setReadOnly(true);
  }
}
```

3.3.4.11.2. CheckBox标签属性

可参考TextField的标签属性。

3.3.4.11.3. CheckBox对象事件

表 3.15.

事件名	用途
blur	失去焦点时触发的事件。

事件名	用途
change	选项的选择状态发生改变时触发的事件。
click	点击选项按钮时触发的事件。
focus	获得焦点时触发的事件。
mouseover	鼠标移到组件上时触发的事件。
mouseout	鼠标移出组件时触发的事件。

3.3.5. Tab组件

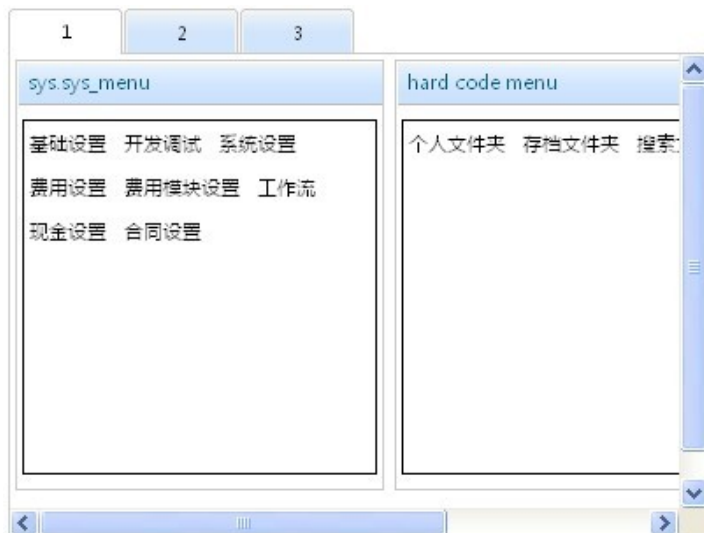
Tab组件是一个标签容器组件，每个标签对应一个页面。

3.3.5.1. Tab定义

在screen文件中我们通过<a:tabPanel>标签来定义tab的一个对象容器，用<a:tab>标签定义每一个tab对象。

```
<a:tabPanel height="300" width="400">
  <a:tabs>
    <a:tab prompt="1" selected="true">
      <a:hBox>
        <a:form id="window" title="sys.sys_menu">
          <div style="height:200px;width:200px;border:1px solid #000000">
          </div>
        </a:form>
        <a:form id="window2" title="hard code menu">
          <div style="height:200px;width:300px;border:1px solid #000000">
          </div>
        </a:form>
      </a:hBox>
    </a:tab>
    <a:tab prompt="2" ref="datepicker.screen"/>
    <a:tab prompt="3"><span>tab demo!</span></a:tab>
  </a:tabs>
</a:tabPanel>
```

生成的界面如下：



tabPanel标签下定义了tabs标签，tabs标签中每个tab标签即一个标签页。tab标签的属性prompt定义了标签的标题，selected属性定义了标签页是否被选中。

tab标签下可以直接编写screen标签代码，也可以通过属性ref指定引用screen页面的路径，引用页面是延迟加载的，只有当页面被选中时，才会首次加载。

3.3.5.2. Tab标签属性

表 3.16.

属性名	用途	默认值	是否必填
bodyClassName	标签容器的样式表。		
bodyStyle	标签容器的样式。		
prompt	标签显示的标题。		
ref	标签引用页的screen文件。		
selected	标签页是否被选中。	false	
closeable	标签是否可关闭。	false	
disabled	标签是否不可用。	false	
tabClassName	标签的样式表。		
tabStyle	标签的样式		

3.3.6. Tree组件

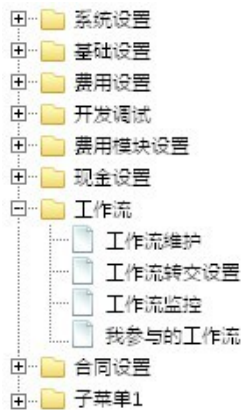
Tree组件是一个树状导航组件。

3.3.6.1. Tab定义

在screen文件中我们通过<a:tree>标签来定义一个tree对象。

```
<a:tree id="functionTree" bindTarget="function_tree_ds" displayField="function_name"
height="400" idField="function_id" parentField="parent_function_id" showCheckBox="false"
expandField="expanded" style="margin:5px;" width="300"/>
```

生成的界面如下:



树中的每一个节点都是对应所绑定的数据集(dataset)中每一条记录(record)。

节点与节点之间的层次关系是通过tree标签的属性idField和parentField来联系的，idField属性指定的是record中用来表示该节点的唯一标识(ID)的字段(field)名，parentField属性指定的record中用来表示该节点的父节点的ID的field名。

tree标签的属性displayField指定的record中用来表示节点文本信息的field名。

3.3.6.2. 树节点渲染

渲染函数(renderer)

很多情况下我们并不需要直接显示数据，而是需要对数据进行一下加工，例如我们需要将每个节点变成一个可以导航页面的链接。这个时候我们就需要在tree标签上指定renderer渲染函数来实现。

首先我们需要实现一个renderer函数，通过取得节点用来表示url的field值，返回一段外层套有a标签的html代码。

```
function linkrenderer(text, record, node){
    var url = record.get('command_line');
    if(url){
        return '<a target="main" href="'+url+'">'+text+'</a>';
    }else{
        return text;
    }
}
```

接下来在tree标签上指定renderer属性为linkrenderer.

```
<a:tree renderer="linkrenderer"/>
```

3.3.6.3. Tree标签属性

表 3.17.

属性名	用途	默认值	是否必填
checkField	record中用来显示树节点是否被选中的field。只有在属性showCheckBox为true的情况下才有效果。	checked	
displayField	record中用来显示树节点文本信息的field。	name	
expandField	record中用来表示树节点是否被伸展的field。	expanded	
idField	record中用来指定树节点唯一标识的field。	id	
parentField	record中用来指定树节点的父节点唯一标识的field。	pid	
renderer	节点渲染器		
sequenceField	record中用来指定树节点序号的field。	sequence	
showCheckBox	是否显示多选框	false	

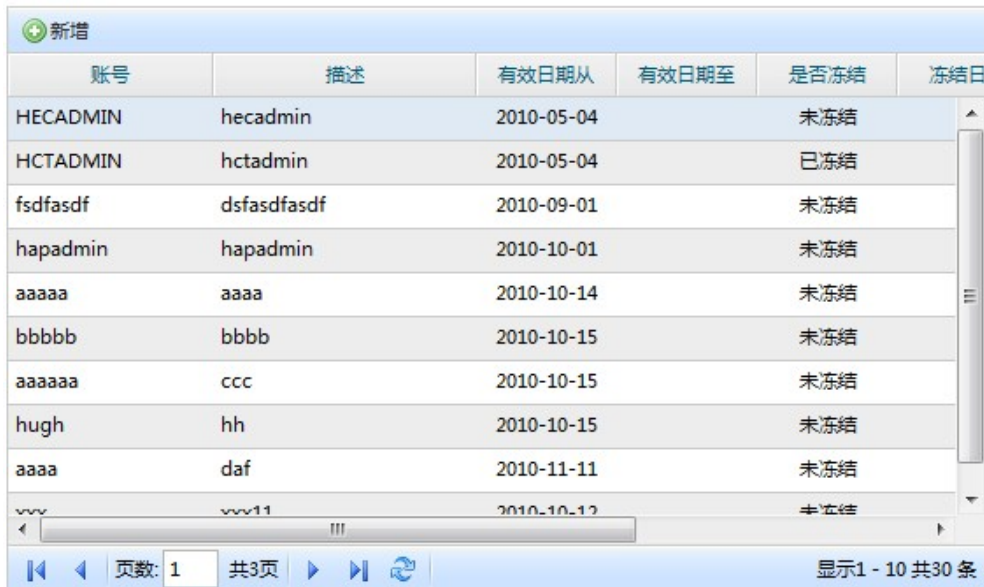
3.3.7. Grid组件

Grid组件是APL中非常重要的一个数据UI组件, 它支持锁定, 复合表头, 排序等特性.

我们通过<a:grid>标签来定义一个grid对象.

```
<a:grid bindTarget="sys_user_result_ds" navBar="true" width="800" id="sys_user_define_grid" height="400">
  <a:toolBar>
    <a:button icon="../../images/add.gif" text="HAP_NEW" click="addUserInfo"/>
  </a:toolBar>
  <a:columns>
    <a:column width="117" name="user_name" sortable="true"/>
    <a:column width="150" name="description" sortable="true"/>
    <a:column width="80" align="center" name="start_date" renderer="Aurora.formatDate" sortable="true"/>
    <a:column width="80" align="center" name="end_date" renderer="Aurora.formatDate" sortable="true"/>
    <a:column width="80" align="center" name="frozen_flag_display" sortable="true"/>
    <a:column width="80" align="center" name="frozen_date" renderer="Aurora.formatDate" sortable="true"/>
    <a:column width="70" name="edit" align="center" prompt="HAP_EDIT" renderer="editUser"/>
  </a:columns>
</a:grid>
```

生成的界面如下:



账号	描述	有效日期从	有效日期至	是否冻结	冻结日
HECADMIN	hecadmin	2010-05-04		未冻结	
HCTADMIN	hctadmin	2010-05-04		已冻结	
fsdfasdf	dsfasdfasdf	2010-09-01		未冻结	
hapadmin	hapadmin	2010-10-01		未冻结	
aaaaa	aaaa	2010-10-14		未冻结	
bbbbb	bbbb	2010-10-15		未冻结	
aaaaaa	ccc	2010-10-15		未冻结	
hugh	hh	2010-10-15		未冻结	
aaaa	daf	2010-11-11		未冻结	
vvv	vvv11	2010-10-12		未冻结	

页数: 1 共3页 显示1 - 10 共30 条

3.3.7.1. 数据绑定

Grid的数据来源是通过DataSet来获取的, 所以Grid需要绑定到一个DataSet上才能正常显示数据.

在grid上我们配置bindTarget属性来指定grid的数据源, bindTarget的值对应到一个具体的dataset的id.

在grid上我们可以配置多个显示列, 在列上指定name值, 对应到dataset中field的name.

3.3.7.2. 列对齐

column上的align属性, 可以指定为center, left, right, 对应列中的文字排列方式分别为居中, 靠左, 靠右. 默认值是center

```
<a:grid bindTarget="sys_user_result_ds" navBar="true" width="800" id="sys_user_define_grid" height="400">
  ...
  <a:columns>
    <a:column width="117" name="user_name" align="center"/>
    <a:column width="150" name="description"/>
    ...
  </a:columns>
</a:grid>
```

3.3.7.3. 列锁定

column上的lock属性, 可以指定为true或者false, 当指定为true的时候当前列将会被锁定. 默认值是false

```
<a:grid bindTarget="sys_user_result_ds" navBar="true" width="800" id="sys_user_define_grid" height="400">
  ...
  <a:columns>
    <a:column width="117" name="user_name" lock="true"/>
    <a:column width="150" name="description"/>
    ...
  </a:columns>
```

```
</a:grid>
```

3.3.7.4. 调整列宽

column上的resizable属性, 可以指定为true或者false, 当指定为true的时候当前列可以动态改变宽度. 默认值是true

```
<a:grid bindTarget="sys_user_result_ds" navBar="true" width="800" id="sys_user_define_grid" height="400">
  ...
  <a:columns>
    <a:column width="117" name="money" resizable="false"/>
    ...
  </a:columns>
</a:grid>
```

3.3.7.5. 排序

column上的sortable属性, 我们可以指定为true或者false来指定当前列是否可以排序. 默认值是false

注意: 当点击表头栏的时候会进行排序, 这里是通过ajax重新进行了一次查询, 是对所有数据的排序.

```
<a:grid bindTarget="sys_user_result_ds" navBar="true" width="800" id="sys_user_define_grid" height="400">
  ...
  <a:columns>
    <a:column width="117" name="user_name" sortable="true"/>
    ...
  </a:columns>
</a:grid>
```

3.3.7.6. 复合表头

很多情况下我们需要对表头进行组合, 这个时候我们可以通过嵌套column列来实现

注意: 当点击表头栏的时候会进行排序, 这里是通过ajax重新进行了一次查询, 是对所有数据的排序.

```
<a:grid bindTarget="sys_user_result_ds" navBar="true" width="800" id="sys_user_define_grid" height="400">
  ...
  <a:columns>
    <a:column prompt="复合表头">
      <a:column width="117" name="user_name" prompt="列1"/>
      <a:column width="117" name="user_name" prompt="列2"/>
      ...
    </a:column>
    ...
  </a:columns>
</a:grid>
```

显示效果:

+ 新增	
复合表头	
列1	列2
HECADMIN	hecadmin
HCTADMIN	hctadmin

3.3.7.7. 列渲染

渲染函数 (renderer)

很多情况下我们并不需要直接显示数据, 而是需要对数据进行一下加工, 例如对于大于0的金额显示绿色, 小于0的金额显示红色. 这个时候我们就需要 在column上指定renderer渲染函数来实现.

首先我们需要实现一个renderer函数, 来判断当前金额, 然后返回一段html代码.

```
function moneyRenderer(value, record, name) {
  if (value >= 0) {
    return '<font color="green">' + value + '</font>'
  } else {
    return '<font color="red">' + value + '</font>'
  }
}
```

接下来在column上指定renderer属性为moneyRenderer.

```
<a:grid bindTarget="sys_user_result_ds" navBar="true" width="800" id="sys_user_define_grid" height="400">
  ...
  <a:columns>
    <a:column width="117" name="money" renderer="moneyRenderer"/>
    ...
  </a:columns>
</a:grid>
```

3.3.7.8. 汇总列

grid提供了汇总的功能, 我们可以通过在列上配置footerRenderer来实现.

例如: 我们需要对金额列进行汇总. 首先需要实现一个汇总的函数summaryRenderer

```
function summaryRenderer(datas, name) {
  var sum = 0;
  for (var i = 0; i < datas.length; i++) {
    var r = datas[i];
    var d = r.get(name);
    var n = parseFloat(d);
    if (!isNaN(n)) {
      sum += n;
    }
  }
  return '合计金额: <font color="red">' + Aurora.formatNumber(sum) + '</font>';
}
```



```
}

```

接下来在column上指定footerRenderer属性为summaryRenderer.

```
<a:grid bindTarget="sys_user_result_ds" navBar="true" width="800" id="sys_user_define_grid" height="400">
  ...
  <a:columns>
    <a:column width="117" name="money" footerRenderer="summaryRenderer"/>
    ...
  </a:columns>
</a:grid>

```

注意:这里的summaryRenderer的数据范围只能局限在当前页. 无法统计到所有数据. 换句话说无法统计到下一页的数据

3.3.7.9. 工具栏

在Grid组件上提供了一个工具栏, Grid本身提供了内置的几个按钮, 我们还可以自定义新的按钮.

```
<a:grid id="grid" bindTarget="sys_function_result_ds" height="300" navBar="true" width="500">
  <a:toolBar>
    <a:button type="add"/>
    <a:button type="delete"/>
    <a:button type="save"/>
  </a:toolBar>
  <a:columns>
    ...
  </a:columns>
</a:grid>

```

我们可以看到在grid上我们添加了3个按钮. 通过指定type值可以确定这个按钮的逻辑.

Grid组件默认提供了3种类型的按钮.

- add -- 在当前grid中新增一条记录.
- delete -- 删除当前选中的记录.
- save -- 保存修改过的记录.

除了Grid内置的按钮, 我们还可以自定义按钮

```
<a:grid id="grid" bindTarget="sys_function_result_ds" height="300" width="500">
  <a:toolBar>
    <a:button icon="../../../images/add.gif" text="HAP_NEW" click="addUserInfo"/>
  </a:toolBar>
  <a:columns>
    ...
  </a:columns>
</a:grid>

```

icon代表按钮的图标, text表示按钮的文本内容, click指定当前按钮的动作。

3.3.7.10. 编辑器

大部分情况下我们需要对数据进行编辑, APL的Grid组件可以集成编辑器, 只需要在列中配置editor。

```
<a:grid id="grid" bindTarget="sys_function_result_ds" height="300" width="500">
  <a:toolBar>
    <a:button type="add"/>
    <a:button type="delete"/>
    <a:button type="save"/>
  </a:toolBar>
  <a:columns>
    <a:column name="function_code" editor="sys_function_result_grid_tf" width="100"/>
    <a:column name="function_name" editor="sys_function_result_grid_tf" width="120"/>
    <a:column name="parent_function_name" editor="sys_function_result_grid_lv" width="120"/>
    ...
  </a:columns>
  <a:editors>
    <a:textField id="sys_function_result_grid_tf"/>
    <a:lov id="sys_function_result_grid_lv"/>
  </a:editors>
</a:grid>
```

这里我们可以看到Grid中增加了一个editors子标签, 在editors标签下定义了几个编辑器。我们可以把editors理解为grid下的一个组件库。每一个编辑器需要指定一个唯一的id, 然后在列上配置editor指定你所需要的编辑器id。这样当你点击grid的某一个单元格的时候, grid 首先回去找列中对应的editor值, 然后再去组件中根据id来查找, 最后显示到指定的单元格上。

3.3.8. Table组件

Table组件是一个数据UI组件, 类似于Grid组件。

3.3.8.1. Table定义

在screen文件中我们通过<a:table>标签来定义一个table对象。

```
<a:table id="sys_user_define_grid" bindTarget="sys_user_result_ds" percentWidth="90"
  style="margin:7px;" navBar="true" navBarType="simple" title="SYS_USER.USER_SEARCH">
  <a:columns>
    <a:column>
      <a:column name="user_name" footerRenderer="frdr" percentWidth="10"/>
      <a:column name="description" editor="description_tf" percentWidth="20"/>
    </a:column>
    <a:column name="start_date" align="center" footerRenderer="frdr"
      percentWidth="10" renderer="Aurora.formatDate"/>
    <a:column name="end_date" align="center" percentWidth="10" renderer="Aurora.formatDate"/>
    <a:column name="frozen_flag_display" align="center" percentWidth="10"/>
    <a:column name="frozen_date" align="center" editor="frozen_date_table_dp"
      percentWidth="10" renderer="Aurora.formatDate"/>
  </a:columns>
```

```

    <a:column name="assign_role" align="center" percentWidth="10" prompt="SYS_USER.ROLE_ASSIGN"
        renderer="assignRole"/>
    <a:column name="set_password" align="center" percentWidth="10" prompt="MODIFY_PASSWORD"
        renderer="setPassword"/>
    <a:column name="edit" align="center" percentWidth="10" prompt="HAP_EDIT" renderer="editUser"/>
</a:columns>
<a:editors>
    <a:datePicker id="frozen_date_table_dp"/>
    <a:textField id="description_tf"/>
</a:editors>
</a:table>

```

生成的界面如下：

用户查询								
账号	描述	有效日期从	有效日期至	是否冻结	冻结日期	分配角色	修改密码	编辑
HECADMIN	hecadmin	2010-05-04		未冻结		分配角色	修改密码	编辑
HCTADMIN	hctadmin	2010-05-04		已冻结		分配角色	修改密码	编辑
fsdfasdf	dsfasdfasdf	2010-09-01		未冻结		分配角色	修改密码	编辑
hapadmin	hapadmin	2010-10-01		未冻结		分配角色	修改密码	编辑
aaaaa	aaaa	2010-10-14		未冻结		分配角色	修改密码	编辑
bbbbbb	bbbb	2010-10-15		未冻结		分配角色	修改密码	编辑
aaaaaaa	ccc	2010-10-15		未冻结		分配角色	修改密码	编辑
hugh	hh	2010-10-15		未冻结		分配角色	修改密码	编辑
aaaa	daf	2010-11-11		未冻结		分配角色	修改密码	编辑
ab	ab	2011-02-01		未冻结		分配角色	修改密码	编辑
1		1						

3.3.8.2. Table与Grid的异同

表 3.18.

异同项	Grid	Table
没有height属性，高度是自适应高度，随着行数的增加而增高。	×	√
通过percentWidth属性，宽度能设成百分比宽度。	×	√
通过title属性能设置标题。	×	√
有锁定列的功能。	√	×
有调整列宽的功能。	√	×
有排序功能。	√	×
有工具栏toolBar。	√	×

3.3.9. 窗口组件(Window)

APL中提供的窗口组件不同于html中的原生window, 它是通过div模拟出来的. 本质上和父页面是在同一个dom树中.

window组件的内容是通过ajax动态加载进来的. 所以加载页面中的所有对象在父页面中都可以直接获取到, 反过来window也可以直接获取父页面的所有对象.

window组件的创建都是通过JavaScript脚本创建的.

```
function openWindow() {
    var win = new Aurora.Window({id:'mywin', url:'user.screen',title:'窗口', height:400,width:700});
}
```

通过new Aurora.Window我们可以创建一个window窗口. id是窗口的唯一标识, url指定当前窗口需要加载的screen文件. title指定打开窗口的标题, height和width分别指定窗口的大小.

表 3.19.

参数名	用途	默认值	是否必填
id	window窗口的id。		true
title	window窗口的标题。		false
url	window窗口的url。		true
height	window窗口的高度。	400	false
width	window窗口的宽度。	350	false

有时候我们仅仅需要显示一个提示信息, 或者一个简单的警告. 这个时候通过url加载就比较麻烦. APL为我们提供了几个通用的窗口函数.

- 提示信息窗口

```
Aurora.showMessage(title, msg, callback, width, height);
```

表 3.20.

参数名	用途	默认值	是否必填
title	window窗口的标题。		true
msg	需要提示的信息内容		true
callback	回调函数, 不需要的话用null		false
height	window窗口的高度。	100	false
width	window窗口的宽度。	300	false

- 带警告图标的窗口

```
Aurora.showWarningMessage(title, msg, callback, width, height);
```

配置参数参考“提示信息窗口”

- 带信息图标的窗口

```
Aurora.showInfoMessage(title, msg, callback, width, height);
```

配置参数参考“提示信息窗口”

- 带错误图标的窗口

```
Aurora.showErrorMessage(title, msg, callback, width, height);
```

配置参数参考“提示信息窗口”

- 带确定取消按钮的确认窗口

```
Aurora.showConfirm(title, msg, okfun, cancelfun, width, height);
```

表 3.21.

参数名	用途	默认值	是否必填
title	window窗口的标题。		true
msg	需要提示的信息内容		true

参数名	用途	默认值	是否必填
okfun	确定按钮回调函数, 不需要的话用null		
cancelfun	取消按钮回调函数, 不需要的话用null		
height	window窗口的高度。	100	false
width	window窗口的宽度。	300	false

3.3.10. 上传组件

正文 ...

3.3.11. TreeGrid

TreeGrid是一个树状结构的表格组件, 它集成了Grid和Tree的多种特性.

```
<a:treeGrid bindTarget="function_tree_ds" expandField="_expanded" height="400" id="functionTreeGrid"
  idField="function_id" parentField="parent_function_id" showCheckBox="true" width="570">
  <a:columns>
    <a:column name="function_name" prompt="功能名称" width="250"/>
    <a:column name="function_code" prompt="功能代码" width="120"/>
    <a:column editorFunction="expandEditorFunction" name="expanded" prompt="是否展开" width="80"/>
    <a:column align="right" editor="grid_nf" name="sequence" prompt="序列号" width="100"/>
  </a:columns>
  <a:editors>
    <a:numberField id="grid_nf"/>
    <a:checkBox id="grid_cb"/>
  </a:editors>
</a:treeGrid>
```

功能名称	功能代码	是否展开	序列号
系统设置	SYS	<input type="checkbox"/>	1
基础设置	FND	<input checked="" type="checkbox"/>	2
开发调试	DEBUG	<input type="checkbox"/>	3
费用设置	EXP	<input type="checkbox"/>	3
现金设置	CSH	<input type="checkbox"/>	4
费用模块设置	EXPM	<input type="checkbox"/>	4
工作流	WFL	<input type="checkbox"/>	5
工作流维护	WFL2010		1
工作流监控	WFL3010		2
工作流转交设置	WFL2110		2
我参与的工作流	WFL1070		6
合同设置	CON	<input type="checkbox"/>	7
子菜单1	TEST1	<input type="checkbox"/>	10

从上图我们可以看到TreeGrid主要由两部分组成, 左边的Tree结构, 右边的Grid结构. TreeGrid的配置和Grid类似, 也是通过column配置列信息, editors配置所有的编辑器.

和grid不同的是我们要指定TreeGrid的树节点信心. 这里我们通过2个属性idField和parentField来配置. idField代表每一条record的唯一标识, parentField顾名思义代表 每一条record的父节点值(idField). 除了这2个属性我们还可以通过指定expandField属性来确定每个节点是否是展开状态, 以及通过showCheckBox属性确定是否需要显示checkbox.

column的配合和grid相同, 也可以定义renderer, lock等特性. 同样TreeGrid也支持可编辑组件.

表 3.22. TreeGrid标签属性

属性名	用途	默认值	是否必填
checkField	record中用来显示树节点是否被选中的field。只有在属性showCheckBox为true的情况下才有效果。	checked	
displayField	record中用来显示树节点文本信息的field。	name	
expandField	record中用来表示树节点是否被伸展的field。	expanded	
idField	record中用来指定树节点唯一标识的field。	id	
parentField	record中用来指定树节点的父节点唯一标识的field。	pid	
renderer	节点渲染器		
sequenceField	record中用来指定树节点序号的field。	sequence	
showCheckBox	是否显示多选框	false	

3.4. 个性化及定制

个性化及定制:

3.4.1. 界面定制

界面定制及更改:

3.4.2. 组件样式修改

组件样式修改:

3.4.3. 修改网页整体布局

修改网页整体布局:

3.5. 多语言支持

个性化及定制:

3.5.1. 基于数据库存储的多语言支持

基于数据库存储的多语言支持:

3.5.2. 自定义多语言实现

自定义多语言实现:

3.5.3. Screen及模板资源文件中的多语言支持

xxx:

第 4 章 Aurora服务层 (ASL)

4.1. ASL (Aurora Service Layer) 总体架构概述

ASL结构图:

4.2. 业务模型 (Business Model)

Business Model (以下简称BM) 是实体对象 (数据库表, 视图等) 的应用层模型, 通过XML格式的文件进行配置。

4.2.1. BM的创建与使用

4.2.1.1. Business Model的基本属性

一个典型的BM文件如下所示:

```
<bm:model xmlns:bm="http://www.aurora-framework.org/schema/bm" baseTable="EMP" alias="e" >
  <bm:fields>
    <bm:field name="empno" databaseType="BIGINT" dataType="java.lang.Long"/>
    <bm:field name="employee_name" physicalName="ename" databaseType="VARCHAR" dataType="java.lang.String"/>
    <bm:field name="job" databaseType="BIGINT" dataType="java.lang.String"/>
    <bm:field name="mgr" databaseType="BIGINT" dataType="java.lang.Long"/>
    <bm:field name="hiredate" databaseType="DATE" dataType="java.sql.Date"/>
    <bm:field name="deptno" databaseType="BIGINT" dataType="java.lang.Long"/>
    <bm:field name="sal" databaseType="FLOAT" dataType="java.lang.Long"/>
    <bm:field name="comm" databaseType="FLOAT" dataType="java.lang.Long"/>
    <bm:field name="current_date" databaseType="DATE" dataType="java.sql.Date" expression="sysdate"/>
  </bm:fields>
  <bm:primary-key>
    <bm:pk-field name="empno"/>
  </bm:primary-key>
</bm:model>
```

简单地说:

<bm:model>相当于一个表;

<bm:fields>相当于表中的一个字段。

下面将对BM中各种tag及其属性的含义进行说明。

<bm:model>是BM的顶层元素。通过baseTable属性设置其对应的数据库实体 (表或视图) 的名字, alias设置其别名。

上面的例子表示对数据库中名为EMP的表建立模型, 并为它设置别名e。通过这个业务模型来执行查询时, 将会生成

```
select ... from EMP e
```

这样的语句。

<bm:fields>部分定义业务模型中包含的字段。通过name来设置字段的名称，通过databaseType和dataType分别设置字段在数据库中的类型，以及对应的java对象类型。数据库类型和Java类型可以不一致，例如，在数据库中定义为BIGINT类型，在Java中可能希望以String来表示。

注意第二个字段，除了name之外，还定义了physicalName。name表示字段从数据库中读取出来以后，以什么名字来命名，而physicalName表示字段在数据库中实际的字段名。在这个例子中，name与physicalName不一致，实际产生的SQL语句将是这样：

```
select ename as employee_name from EMP
```

name属性必须设置，并且在整个<bm:model>范围内，必须唯一。physicalName如果不设置，则表示和name相同。

<bm:field>不一定对应只对应物理存在的字段，也可以是一个表达式。注意最后一个字段，设置了expression="sysdate"，这将在查询SQL中生成

```
select sysdate as current_date
```

这样的语句。通过expression属性，可以设置任意合乎当前数据库语法规则的表达式。例如，要实现select f1+f2 as f3这样的效果，可以设置<bm:field name="f3" expression="f1+f2" />。

<bm:primary-key>部分设置业务模型对应的表的主键。每个<bm:pk-field>设置一个主键对应的字段，通过name属性设置字段名，这个名字必须和前面<bm:fields>部分所定义的某一个字段名一致。如果是联合主键，就为主键中的每一个字段设置<bm:pk-field>属性。

4.2.1.2. BM的使用

将上述文件存为(Web应用根目录)/WEB-INF/classes/test/emp.bm文件，就可以通过该文件，实现基本的数据库操作。

和Java class的命名规则相似，这个文件被引用的名字为test.emp，从WEB-INF/classes/目录开始，每一级子目录的名字，以"."代替"/"，再拼接上文件名，去掉.bm扩展名，就得到BM的完整名称。

Aurora框架提供了一系列的Java API来使用BM，但在开发应用程序的时候，多半不需要通过Java代码来使用。Aurora提供了一种方便的机制，可以直接将BM发布为服务。客户端通过HTTP方式调用BM服务，以JSON格式来传递输入参数，Server端同样以JSON来返回调用结果。

以前面的例子来说，BM文件保存完毕后，启动Web服务器，直接在浏览器中访问 <http://<主机地址>/web应用目录/autocrud/test.emp/query>，就可以得到类似于下面的结果：

如果将上面地址最后的 /query 换成 /insert，再传递参数：
`_request_data={empno:1,employee_name:"test"}`，即可向数据库中插入一条记录。


这样，就避免了传统了Java建模及O/R Mapping->Java编码->Java方法发布为JSON服务的繁琐过程，通过简单的配置（这个过程在IDE的支持下可实现全自动数据库反向工程），一步即可得到客户端javascript程序所需的JSON格式的数据服务。

BM支持insert, update, delete, query, execute等5种基本操作。对于非insert/update/delete类的操作，如复杂sql语句或存储过程调用，都归为execute类，通常由开发者自行撰写相应的SQL。


Aurora示例程序中的bm.screen工具，是一个BM调试的Web工具，通过浏览器访问该界面，输入BM名称，选择要执行的操作，再输入参数，就可以对BM进行调用，在界面上查看服务端返回的结果。

测试

BM:

方法: 

参数:

分页: ☐ 每页记录: 页号: 自动计数: ☐ 排序:  排序字段:

```
{
  "result": {
    "record": [
      {
        "created_by": 2,
        "frozen_flag": "N",
        "encrypted_foundation_password": "202CB962AC59075B964B07152D234B70",
        "encrypted_user_password": "202CB962AC59075B964B07152D234B70",
        "creation_date": "2011-03-21 00:00:00",
        "user_name": "4444",
        "last_updated_by": 2,
        "end_date": "2011-03-21 00:00:00",
        "description": "333322",
        "user_id": 364,
        "employee_id": 41,
        "start_date": "2011-03-02 00:00:00",
        "last_update_date": "2011-07-01 00:00:00",
        "creation_date": "2011-06-02 00:00:00",
        "frozen_flag": "N",
        "created_by": 2,
        "user_name": "hand",
        "last_updated_by": 2,
        "encrypted_foundation_password": "187CCA46AB69A66CDFF777315459C07C",
        "description": "hand",
        "user_id": 388,
        "encrypted_user_password": "187CCA46AB69A66CDFF777315459C07C",
        "start_date": "2011-06-02 00:00:00",
        "last_update_date": "2011-06-02 00:00:00",
        "creation_date": "2011-06-02 00:00:00"
      }
    ]
  }
}
```

此外，通过Aurora IDE，也可以方便地查看、执行通过BM生成的SQL。这部分内容详见Aurora IDE使用手册。

4.2.2. 通过BM执行查询

4.2.2.1. 字段, 别名与表达式

通过BM生成SQL查询的基本逻辑是：

```
select 字段1, 字段2, 字段3, ... , 字段n from {baseTable} as {alias}
```

其中，如果某个字段定义的名称和physicalName不一致，对此字段将生成select {physicalName} as {name}这样的SQL语句。

如果某个字段希望定义在BM中，但又不希望让该字段出现在select语句中（例如，只用于更新的字段），可以设置该字段的forSelect="false"属性。

如果某个字段不是基于实际存在表字段，而是一段SQL表达式，或者希望对该字段调用SQL函数进行处理，那么可以设置该字段的expression属性。例如，下面的例子在获取creation_date字段时，调用

oracle内建to_char()函数进行格式化:

```
<bm:field name="creation_date" expression="trunc(user_name)" />
```

如果BM可能会用于多种数据库, 请谨慎使用自定义的SQL语句, 以免带来SQL语法兼容性问题。

4.2.2.2. SQL Join

实际应用中我们经常需要将多个表join在一起查询。在BM中, 通过配置基表与其它表的关系, 可实现这个需求。

在以下的例子中, 我们假设有一个dept表存储公司中的部门, emp表存储公司中的员工, 每个员工属于一个特定的部门。首先, 对部门表建立BM:

```
<bm:model xmlns:bm="http://www.aurora-framework.org/schema/bm" baseTable="dept" >
  <bm:fields>
    <bm:field name="deptno" databaseType="BIGINT" dataType="java.lang.Long"/>
    <bm:field name="dname" databaseType="VARCHAR" dataType="java.lang.String"/>
  </bm:fields>
  <bm:primary-key>
    <bm:pk-field name="empno"/>
  </bm:primary-key>
</bm:model>
```

然后, 对前面一节的例子进行扩展, 建立emp与dept表的关联关系, 并在emp表中引用

```
<?xml version="1.0" encoding="UTF-8"?>
<bm:model xmlns:bm="http://www.aurora-framework.org/schema/bm" alias="e" baseTable="EMP" needAccessControl="false">
  <bm:fields>
    <bm:field name="empno" databaseType="BIGINT" datatype="java.lang.Long"/>
    <bm:field name="employee_name" databaseType="VARCHAR" datatype="java.lang.String" physicalName="ename"/>
    <bm:field name="job" databaseType="BIGINT" datatype="java.lang.String"/>
    <bm:field name="mgr" databaseType="BIGINT" datatype="java.lang.Long"/>
    <bm:field name="hiredate" databaseType="DATE" datatype="java.sql.Date"/>
    <bm:field name="deptno" databaseType="BIGINT" datatype="java.lang.Long"/>
    <bm:field name="sal" databaseType="FLOAT" datatype="java.lang.Long"/>
    <bm:field name="comm" databaseType="FLOAT" datatype="java.lang.Long"/>
  </bm:fields>
  <bm:primary-key>
    <bm:pk-field name="empno"/>
  </bm:primary-key>
  <bm:relations>
    <bm:relation name="d" joinType="LEFT OUTER" refModel="test.dept">
      <bm:reference foreignField="deptno" localField="deptno"/>
    </bm:relation>
    <bm:relation name="m" joinType="LEFT OUTER" refModel="test.emp">
      <bm:reference expression="e.mgr = m.empno and m.deptno is not null"/>
    </bm:relation>
  </bm:relations>
  <bm:ref-fields>
    <bm:ref-field name="department_name" relationName="d" sourceField="dname"/>
    <bm:ref-field name="manager_name" relationName="m" sourceField="employee_name"/>
  </bm:ref-fields>
  <bm:data-filters>
```

```

    <bm:data-filter enforceOperations="query" expression="e.hiredate is not null"/>
  </bm:data-filters>
  <bm:query-fields>
    <bm:query-field field="empno" queryOperator="="/>
    <bm:query-field field="deptno" queryOperator="="/>
    <bm:query-field field="employee_name" queryOperator="like"/>
    <bm:query-field name="hiredate_from" dataType="java.sql.Date" queryExpression="e.hiredate >= ${@hiredate_from}"/>
    <bm:query-field name="hiredate_to" dataType="java.sql.Date" queryExpression="e.hiredate <= ${@hiredate_to}"/>
  </bm:query-fields>
</bm:model>

```

在上例中，<relations>部分配置BM与其他BM的JOIN关系。通过relation标签定义一个与其他BM的关联，在下面通过1..n个reference标签定义关联的条件。上例中第一个关联BM：

```

<bm:relation name="d" joinType="LEFT OUTER" refModel="test.dept">
  <bm:reference foreignField="deptno" localField="deptno"/>
</bm:relation>

```

表示与名为test.dept的BM进行left outer join，通过下面的reference标签，设置以本BM中的deptno字段（由localField属性指定）等于test.dept的deptno字段（由foreignField属性指定），来构造join条件。<relation>的name属性，是对此关联关系的命名，在别的地方将通过这个名字来引用，也相当于被关联的表的别名。这样，形成的sql语句将是：

```

FROM EMP e
LEFT OUTER JOIN dept d ON e.deptno = d.deptno

```

如果有多个关联字段（例如联合主键），就设置多个<reference>标签，逐一设置localField和foreignField属性。如果关联条件比较复杂，不是简单的A表某字段=B表某字段这样的形式，可以通过reference的expression属性来设置join的SQL表达式，如第二个reference：

```

<bm:relation name="m" joinType="LEFT OUTER" refModel="test.emp">
  <bm:reference expression="e.mgr = m.empno and m.deptno is not null"/>
</bm:relation>

```

这里，EMP表又与自己进行left outer join，join条件由expression属性设定。这将构成如下SQL：

```

FROM EMP e
LEFT OUTER JOIN EMP m ON e.mgr = m.empno and m.deptno is not null

```

join条件设置好之后，就可以在<ref-fields>部分设置要引用的关联表中的字段。上面的例子中：

```

<bm:ref-fields>
  <bm:ref-field name="department_name" relationName="d" sourceField="dname"/>
  <bm:ref-field name="manager_name" relationName="m" sourceField="employee_name"/>
</bm:ref-fields>

```

我们将员工所在部门的名称和上级主管的名称取出来，并给它们分别命名为 department_name 和 manager_name。其中，ref-field 的 relationName 属性为要引用的表在前面所设置的 relation 的名称，sourceField 为要引用的字段在关联的 BM 中的字段命名。

注意：同一个 BM 中，所有字段，包括基本字段和引用的字段，其 name 属性必须唯一。如果基表中已经有了一个名叫 name 的字段，再引用其他表中也叫 name 的字段，就必须通过 name 属性为这个引用字段起一个新的不重复的名字。

经过上述步骤，设置好关联的 BM 和引用的字段之后，最终形成的 SQL 语句将是：

```
SELECT e.empno, e.ename AS employee_name, e.job, e.mgr, e.hiredate, e.deptno, e.sal, e.comm, d.dname AS department_name, m.
FROM EMP e
LEFT OUTER JOIN dept d ON e.deptno = d.deptno
LEFT OUTER JOIN EMP m ON e.mgr = m.empno and m.deptno is not null
```

4.2.2.3. 查询

4.2.2.3.1. 可选查询条件

通常，一个典型的查询界面会列出一些字段，让用户输入条件，然后根据用户输入，列出符合条件的结果。BM 的 <query-fields> 部分可以定义哪些字段是可用于查询的，以及查询条件是什么。在本例中：

```
<bm:query-fields>
  <bm:query-field field="empno" queryOperator="="/>
  <bm:query-field field="deptno" queryOperator="="/>
  <bm:query-field field="employee_name" queryOperator="like"/>
  <bm:query-field name="hiredate_from" dataType="java.sql.Date" queryExpression="e.hiredate >= {@hiredate_from}"/>
  <bm:query-field name="hiredate_to" dataType="java.sql.Date" queryExpression="e.hiredate <= {@hiredate_to}"/>
</bm:query-fields>
```

这里定义了 5 个查询字段。以第二个为例，指定字段 deptno 为查询参数，查询操作符为 "="，即查询条件为 deptno=（用户输入的值）。为测试这个查询字段的效果，我们在浏览器中输入：

[Web 目录]/autocrud/test.emp_for_query/query?deptno=10

可以看到，返回的查询结果是只包含 deptno=10 的 emp 记录：

```
{"result":{"record":[{"hiredate":"2011-07-26 00:00:00","comm":100,"empno":110,"job":"MANAGER","deptno":10,"manager_name":
```

如果再加上 hiredate_from 参数：

[Web 目录]autocrud/test.emp_for_query/query?deptno=10&hiredate_from=2011-1-1

可以看到，返回结果进一步筛选，只返回deptno=10，并且hiredate在2011-1-1之前的记录。

```
{"result":{"record":[{"hiredate":"2011-07-26 00:00:00","comm":100,"empno":110,"job":"MANAGER","deptno":10,"manager_name":
```

每个<query-field>定义一个可以查询的字段。如果该字段是BM中已经定义过的字段，就设置field属性为字段的名称，例如<query-field field="deptno" />。如果查询参数的名字并不直接对应BM中已定义的字段，如上例中，我们要对hiredate字段进行区间查询，可能会输入hiredate_from和hiredate_to两个值，分别代表区间的上下限，而BM中并没有一个叫hiredate_from的字段。这时，就直接设置query-field的name属性。如果该参数不是字符串类型，还应设置datatype属性。

如果查询条件是简单的字段+操作符模式，可以设置query-field的queryOperator="操作符"。例如，queryOperator="="，代表字段="输入的值"；queryOperator=">="，代表字段>=输入的值。

如果查询条件比较复杂，不能简单地用某字段=某值的方式来表达，可以用queryExpression来设定任意的SQL表达式或子查询语句。例如：

```
<bm:query-field name="hiredate_from" datatype="java.sql.Date" queryExpression="e.hiredate > ${@hiredate_from}"/>
```

这个配置表示，如果用户提交参数中包含hiredate_from，就拼接 "hiredate > (用户输入的hiredate值)" 这样一个查询条件。其中，\${@hiredate_from} 表示用户输入的名为hiredate_from的参数值，其完整XPath路径是\${/parameter/@hiredate_from}，对于SQL查询操作来说，缺省的传入参数是/parameter，所以可以用相对路径@hiredate_from来表示。在queryExpression中，如需使用基表字段，应注意使用别名，以避免多表关联时不同表存在同名字段，而引起的SQL错误。

再举一个自定义查询字段的例子，假设希望查到是管理者的员工，而管理者就是至少有一个员工的直接上级是其本人的员工。这个查询无法直接通过对基表字段的筛选来完成，需要使用SQL子查询，例如：

```
select ... from emp e where exists ( select 1 from emp e1 where e1.mgr = e.empno )
```

那么，我们可以定义一个名为is_manager的参数，如果这个参数传递了值，就在构造上述子查询条件：

```
<bm:query-field name="is_manager" queryExpression="exists ( select 1 from emp e1 where e1.mgr = e.empno )"/>
```

4.2.2.3.2. 数据限制条件

很多时候，我们需要对查询SQL添加一些限制性的条件。例如，当前用户只能看他所在的部门的员工，或者只列出状态不等于作废的订单。不论用户输入什么查询参数，这些限制条件都是必须加上的。这时，我们可以配置BM的<data-filters>标记，添加必需的数据筛选条件：

```
<bm:data-filters>
```

```
<bm:data-filter enforceOperations="query" expression="e.hiredate is not null"/>
</bm:data-filters>
```

这表示，在执行查询的时候，e.hiredate is not null这个条件将会被无条件地加上。如果我们传递一个deptno=10的查询条件，可以看到对该BM执行query操作时候，实际拼接的SQL语句是：

```
SELECT e.empno, e.ename AS employee_name, e.job, e.mgr, e.hiredate, e.deptno, e.sal, e.comm, d.dname AS dname
FROM EMP e
      LEFT OUTER JOIN dept d ON e.deptno = d.deptno
      LEFT OUTER JOIN EMP m ON e.mgr = m.empno and m.deptno is not null
WHERE e.hiredate is not null AND e.deptno = ?
```

如果有多个限制条件，可以设置多个data-filter标记，每一个data-filter设置的expression属性会用and连接起来，再与可选查询条件做and连接，形成最终的where从句。

缺省状况下，限制条件会被拼接到所有可包含where从句的SQL操作中，包括update，delete。这样可以实现比较严密的记录级权限控制。例如，要求用户只能操作自己创建的订单，可以设置created_by=\${/session/@user_id}。即使用户执行update操作，提交了一个别人创建的订单号，后台所执行的SQL语句将类似于：

```
update ord_order set .... where order_id=${/parameter/@order_id} and created_by=${/session/@user_id}
```

由于where从句的表达式不成立（其他人创建的订单，created_by不等于当前用户id），最终也不会有记录被更新。

如果希望某些限制条件只在特定操作时执行，例如只在select中出现，不在update中出现，或者只在select，update中出现，不在delete中出现，那么可以用enforceOperations属性，设定该data-filter希望被使用的操作，用逗号分隔每一个操作，如：enforceOperations="query,update"

4.2.2.3.3. 分页控制

在以autocrud方式执行BM的查询时，可通过传递特定的参数，来控制查询结果是否要分页，以及分页的配置

表 4.1. 分页控制参数

参数	含义
_fetchall	是否返回所有记录，缺省为false，即启用分页。如果设置为true，将一次向客户端传递查询结果中所有的记录。
pagesize	每页记录数
pagenum	返回第几页
_autocount	是否自动计算记录总数。如果设置为true，将自动执行select count(1) from (实际查询语句)，并将结果放置到totalCount属性中。Aurora标准分页控件需要总记录数来显示分页相关的属性。

例如，在浏览器中输入以下地址：


```
[Web目录]/autocrud/test.emp_for_query/query?pagenum=2&pagesize=3&_autocount=true
```

即表示，按每页3条记录，返回第2页的数据，并自动统计记录总数。返回结果如下：

```
{
  "result": {
    "record": [
      {
        "hiredate": "1981-02-22 00:00:00",
        "comm": 500,
        "empno": 7521,
        "job": "SA"
      },
      {
        "hiredate": "2007-09-01 00:00:00",
        "comm": 1400,
        "empno": 7654,
        "job": "SA"
      },
      {
        "hiredate": "1981-09-08 00:00:00",
        "comm": 0,
        "empno": 7844,
        "job": "SA"
      }
    ],
    "totalCount": 17
  },
  "success": true
}
```

4.2.2.3.4. 排序

以下参数可以控制查询结果的排序：

表 4.2. 排序控制参数

参数	含义
ORDER_FIELD	用于排序的字段，与
ORDER_TYPE	排序的顺序，可以是asc或desc，必须与ORDER_FIELD同时时候
_ORDER_FIELD_PARAM_NAME	用于传递排序字段名称的参数名。如果不能用缺省的ORDER_FIELD来命名排序字段，可以通过这个字段来设置排序字段参数的实际名称。例如，设置_ORDER_FIELD_PARAM_NAME=my_order，然后设置my_order=ename，即表示按照ename字段排序。
_ORDER_TYPE_PARAM_NAME	用于传递排序顺序的参数名，如果不能用缺省的ORDER_TYPE来命名排序方式，可以通过这个字段来设置排序方式参数的实际名称。

例如，在浏览器中输入以下地址：

```
[Web目录]/autocrud/test.emp_for_query/query?ORDER_FIELD=employee_name&ORDER_TYPE=desc
```

查看执行日志，生成的SQL将是：

```
SELECT e.empno,e.ename AS employee_name,e.job,e.mgr,e.hiredate,e.deptno,e.sal,e.comm,d.dname
FROM EMP e
      LEFT OUTER JOIN dept d ON e.deptno = d.deptno
      LEFT OUTER JOIN EMP m ON e.mgr = m.empno and m.deptno is not null
WHERE e.hiredate is not null
ORDER BY ename desc
```

此外，也可以在BM上设置defaultOrderBy属性，如果客户端没有传递排序相关的参数，就使用这个属性设置的表达式，来构造order by部分。例如：

```
<bm:model defaultOrderBy="deptno asc,employee_name desc">
...
</bm:model>
```

这样，如果直接访问该BM的/query方法，生成的SQL将包含：ORDER BY deptno asc,employee_name desc

4.2.2.4. 自定义查询语句

如果BM自动生成的查询SQL无法满足实际需求，也可以使用自定义SQL语句来实现查询。在下面的例子中，我们使用Oracle数据库特有的group by rollup语法，来实现对员工数据按部门，职务的分类汇总。

```
<bm:model xmlns:bm="http://www.aurora-framework.org/schema/bm" needAccessControl="false">
  <bm:operations>
    <bm:operation name="query">
      <bm:query-sql>
        select
          decode(grouping(e.job),0,d.dname,1,d.dname||' Total:') as department_name,
          e.job,
          count(e.empno) as total_employee,
          sum(e.sal) as total_salary
        from   emp e, dept d
        #WHERE_CLAUSE#
        group by   rollup(d.dname, e.job)
        #ORDER_BY_CLAUSE#
      </bm:query-sql>
    </bm:operation>
  </bm:operations>
  <bm:fields>
    <bm:field name="department_name"/>
    <bm:field name="job"/>
    <bm:field name="total_employee" datatype="java.lang.Long"/>
    <bm:field name="total_salary" datatype="java.lang.Long"/>
  </bm:fields>
  <bm:data-filters>
    <bm:data-filter expression="e.deptno = d.deptno"/>
  </bm:data-filters>
  <bm:query-fields>
    <bm:query-field name="deptno" queryExpression="d.deptno=${@deptno}" />
    <bm:query-field name="job" queryExpression="e.job=${@job}" />
  </bm:query-fields>
</bm:model>
```

在operations部分，定义一个name="query"的operation，在其下用query-sql标记定义查询SQL。这样，在调用BM执行查询时，由于有名为query的operation存在，Aurora框架将会用query-sql中定义的SQL语句，替代框架自动生成的SQL。

自定义查询语句的BM通常也需要设置fields部分，以定义查询结果集的每一个字段的名称、数据类型。如果没有定义fields部分，Aurora会根据SQL查询生成的结果集的实际结构，来读取数据，这样就无

法对字段数据类型及名称的精确控制。例如，对于SQL结果集中的数字字段，用缺省方式获取的可能是java.math.BigDecimal对象。如果希望得到的是java.lang.Long，就必须像上例一样，明确定义一个field，设置其datatype属性。

4.2.2.4.1. 自定义查询语句的查询条件

和常规BM一样，自定义SQL也可以定义query-fields及data-filters，来使用可选查询参数和必须的限制条件。所不同的是，如果使用了这些设置，在SQL语句中不能直接出现where部分，而是要用#WHERE_CLAUSE#来标记SQL语句中准备放入where从句的位置。Aurora将根据data-filters和query-fields的设置，先后添加必需的数据筛选条件，以及可选的查询条件，自动生成where部分，再替代#WHERE_CLAUSE#。

4.2.2.4.2. 自定义查询语句的排序

在自定义SQL中，如果希望能够像常规BM一样，根据客户端传递的参数来动态排序，就在SQL语句中使用#ORDER_BY_CLAUSE#来标记order by从句放置的地方。如果查询参数中传递了ORDER_FIELD, ORDER_TYPE字段，或者设置了BM的defaultOrderBy属性，Aurora将会自动生成order by从句，替换#ORDER_BY_CLAUSE#标记。

如果访问地址：

```
[Web目录]/autocrud/test.emp_summary/query?deptno=10&ORDER_FIELD=total_employee&ORDER_TYPE=asc
```

生成的SQL将是：

```
select
  decode(grouping(e.job), 0, d.dname, 1, d.dname || ' Total:') as department_name,
  e.job,
  count(e.empno) as total_employee,
  sum(e.sal) as total_salary
from   emp e, dept d
WHERE  e.deptno = d.deptno AND d.deptno=?
group by   rollup(d.dname, e.job)
ORDER BY  total_employee asc
```

其中，#WHERE_CLAUSE#和#ORDER_BY_CLAUSE#经过框架自动处理，将被替代为实际的SQL语句。

4.2.3. 通过BM执行DML

除了查询，BM也能完成insert, update, delete等常规的DML操作。有两种方式：

1. autocrud模式，与查询类似，客户端以json方式提交参数到“[Web目录]/autocrud/[BM名称]/[操作名]”地址，调用BM完成某一种操作。
2. service模式，通过一个svc文件定义一系列要执行的操作，再将结果以JSON或web service形式返回给客户端。

下面将首先以autocrud模式为例，介绍通过BM执行数据操作的基本概念，然后再介绍service模式。

4.2.3.1. 通过BM执行insert

以前面的emp.bm为例，通过BM测试工具，向autocrud/test.emp/insert传递一条JSON数据：

```
{empno:1,employee_name:"test_employee",deptno:10}
```

执行之后，可以看到服务端返回的结果：

```
{"result":{"empno":1,"__parameter_parsed__":true,"deptno":10,"employee_name":"test_employee"},"success":true}
```

查看数据库中的emp表记录，可以看到新建了一条empno等于1的记录：

```
SQL> select empno,ename,deptno from emp where empno=1;
```

EMPNO	ENAME	DEPTNO
1	test_employee	10

查看执行日志，可以看到实际执行的SQL：

```
2011-08-03 16:15:16.930 [aurora.database] [CONFIG]
===== BEGIN [Insert] SQL Statement execution dump =====
INSERT INTO EMP ( empno,ename, job,mgr,hiredate,deptno,sal,comm) VALUES ( ?,?,?,?, ?, ?, ?, ?)
-----Binding info-----
No.1 Access path:@empno Data type of passed value :java.lang.Long Value:1 Output:false Database Type:null
No.2 Access path:@employee_name Data type of passed value :java.lang.String Value:test_employee Output:false Database Type:
No.3 Access path:@job Data type of passed value :[null] Value:null Output:false Database Type:null
No.4 Access path:@mgr Data type of passed value :[null] Value:null Output:false Database Type:null
No.5 Access path:@hiredate Data type of passed value :[null] Value:null Output:false Database Type:null
No.6 Access path:@deptno Data type of passed value :java.lang.Long Value:10 Output:false Database Type:null
No.7 Access path:@sal Data type of passed value :[null] Value:null Output:false Database Type:null
No.8 Access path:@comm Data type of passed value :[null] Value:null Output:false Database Type:null
```

Aurora根据BM中定义的字段，自动拼接insert SQL语句，并将客户端传递过来的参数绑定到SQL语句中，完成insert操作。

以下参数可以对insert语句进行更充分地控制：

4.2.3.1.1. 字段控制

如果希望某个field不出现在insert语句中，可以设置field的forInsert="false"。

4.2.3.1.2. 参数来源

缺省设置下，参与insert操作的字段，其数据来自于数据池的/parameter部分，与该字段同名的参数，对应的XPath路径为/parameter/@[字段名]。如果某个字段的参数来自于其他路径，可以通过parameterPath属性，设置期望的参数路径。例如：

```
<bm:field name="created_by" parameterPath="/session/@user_id" />
```

4.2.3.1.3. 自定义参数取值表达式

如果某字段的insert参数是一个复杂的SQL表达式，而不是简单地来源于某个路径下的参数，那么可以通过insertExpression属性来设置需要实用的SQL表达式，例如：

```
<bm:field name="creation_date" insertExpression="trunc(sysdate)" />
```

在insertExpression中，也可以使用\${}标记，来引用其他的参数。

例1：计算字段

```
<bm:field name="total_price" insertExpression=" ${@total_amount} * ${@unit_price} * my_pkg.get_discount( ${/session/@user_
```

例2：子查询

```
<bm:field name="empno" insertExpression="(select max(empno)+1 from emp)"/>
```

4.2.3.2. 通过BM执行update

update操作的执行与insert类似。我们用BM测试工具向autocrud/test.emp/update传递如下JSON参数：

```
{empno:1, employee_name:"test_employee_update", deptno:20}
```

可以看到，数据库中的记录已被更新：

```
SQL> select empno,ename,deptno from emp where empno=1;
```

EMPNO	ENAME	DEPTNO
1	test_employee_update	20

查看执行日志，对应的SQL操作是：

```
2011-08-03 17:07:54.180 [aurora.database] [CONFIG]
===== BEGIN [Update] SQL Statement execution dump =====
UPDATE EMP e
SET e.ename=?, e.deptno=?
WHERE e.empno = ?
-----Binding info-----
No.1 Access path:@employee_name Data type of passed value :java.lang.String Value:test_employee_update Output:false Database Type:null
No.2 Access path:@deptno Data type of passed value :java.lang.Long Value:20 Output:false Database Type:null
No.3 Access path:@empno Data type of passed value :java.lang.Long Value:1 Output:false Database Type:null
```

与insert不同的是，根据BM中的primary-key设置，update操作会自动拼接[主键字段]=[参数值]这样的where条件，以确保一次操作只更新一条记录。相应的，客户端执行update操作时，提交的参数也必

需包含所有的主键字段。

update操作所涉及的字段参数，与insert类似：

表 4.3. update操作的field控制参数

参数	含义
forUpdate	该字段是否会用于update操作中
updateExpression	执行update操作所用的SQL表达式

4.2.3.2.1. update字段控制

BM在执行update操作时，只更新客户端提交的字段，参数中没有提交的字段就不会被拼接到update语句中。在上面的例子中，就只有ename和deptno字段用于update语句。

而有些字段是不依赖于客户端提交的参数的。例如，last_update_date字段记录最近一次更新操作的时间。无论客户端提交什么样的参数，执行update时都需要更新这个字段。这时，就需要在BM中设置该字段的forceUpdate属性为true。例如：

```
<bm:field name="last_update_date" forceUpdate="true" updateExpression="sysdate" />
```

4.2.3.2.2. 自定义update语句

有时候，某字段的update语句是一段SQL表达式，这段表达式所依赖的输入参数的名称不一定和字段相同。例如：

```
<bm:field name="total_amount" updateExpression="my_pkg.get_sum(${/parameter/@order_head_id})" />
```

按照前面的逻辑，如果没有传递一个名叫total_amount的参数，这个字段就不会被更新。这时，就需要设置一下inputPath属性，设置成该字段所依赖的一个输入参数，如inputPath="/parameter/@order_head_id"。这样，只要传递了order_head_id参数，这个字段就会按updateExpression指定的表达式去执行update。

4.2.3.3. 通过BM执行删除

删除操作相对比较简单。向autocrud/test.emp/delete传递JSON参数：

```
{empno:1}
```

查看执行日志，对应的SQL操作是：

```
2011-08-03 17:36:49.96 [aurora.database] [CONFIG]
===== BEGIN [Delete] SQL Statement execution dump =====
DELETE FROM EMP t
WHERE t.empno = ?
-----Binding info-----
```

```
No.1 Access path:@empno Data type of passed value :java.lang.Long Value:1 Output:false Database Type:null

===== END [Delete] SQL Statement execution dump =====
```

4.2.3.4. 批量操作

如果希望在一次交互中对多条记录进行操作，可以使用批量操作的功能。在autocrud模式下，客户端提交的参数应是一个JSON对象数组，其中的每个JSON对象代表一条需要处理的记录，并用一个状态字段（缺省名为_status）标识要对该记录进行的操作。例如：

```
[
  {deptno:1, _status:"insert", dname:"test_dept_1"},
  {deptno:2, _status:"insert", dname:"test_dept_2"}
]
```

这将提交两条记录，其中的_status属性值都是insert，这表示要对记录执行插入操作。查看执行日志，可以看到针对这两条记录，执行了两次SQL：

```
2011-08-04 14:59:49.300 [aurora.database] [CONFIG] ===== Running model batch update with data from path /parameter,
2011-08-04 14:59:49.300 [aurora.database] [CONFIG] execute insert on record No.0 for model test.dept
2011-08-04 14:59:49.657 [aurora.database] [CONFIG]
...
===== BEGIN [Insert] SQL Statement execution dump =====
INSERT INTO dept ( deptno,dname) VALUES ( ?,?)
-----Binding info-----
No.1 Access path:@deptno Data type of passed value :java.lang.Long Value:1 Output:false Database Type:null
No.2 Access path:@dname Data type of passed value :java.lang.String Value:test_dept_1 Output:false Database Type:null

===== END [Insert] SQL Statement execution dump =====

2011-08-04 14:59:49.658 [aurora.database] [CONFIG] execute insert on record No.1 for model test.dept
...
===== BEGIN [Insert] SQL Statement execution dump =====
INSERT INTO dept ( deptno,dname) VALUES ( ?,?)
-----Binding info-----
No.1 Access path:@deptno Data type of passed value :java.lang.Long Value:2 Output:false Database Type:null
No.2 Access path:@dname Data type of passed value :java.lang.String Value:test_dept_2 Output:false Database Type:null

===== END [Insert] SQL Statement execution dump =====

2011-08-04 14:59:49.674 [aurora.database] [CONFIG] ===== End of batch update for /parameter
```

在后台数据库中，相应地插入了两条记录：

```
SQL> select deptno, dname from dept where dname like 'test%';

DEPTNO DNAME
-----
      1 test_dept_1
      2 test_dept_2
```

也可以混合各种操作，例如：

```
[
{deptno:1,_status:"update",dname:"test1_updated"},
{deptno:2,_status:"delete"}
]
```

这将删掉deptno=2的记录，并更新deptno=1的dname字段。

4.2.3.5. 级联操作

实际应用中，经常有头行结构的数据一次提交保存的需求。BM为这种操作模式提供了级联操作的支持。例如：

```
<bm:model xmlns:bm="http://www.aurora-framework.org/schema/bm" baseTable="dept" needAccessControl="false">
  <bm:fields>
    <bm:field name="deptno" dataType="java.lang.Long" databaseType="BIGINT"/>
    <bm:field name="dname" dataType="java.lang.String" databaseType="VARCHAR"/>
  </bm:fields>
  <bm:primary-key>
    <bm:pk-field name="deptno"/>
  </bm:primary-key>
  <bm:cascade-operations>
    <bm:cascade-operation inputPath="employees" model="test.emp" operations="insert,update,delete"/>
  </bm:cascade-operations>
</bm:model>
```

我们对该BM执行batch_update操作，并提交下面的数据：

```
[
{deptno:3,_status:"insert",dname:"test3_created",employees:[{_status:"insert",employee_name:"emp_test_3.1",deptno:3},{_sta
{deptno:1,_status:"update",dname:"test1_updated",employees:[{_status:"insert",employee_name:"emp_test_1.1",deptno:1},{_sta
}]
```

除了dept自己的参数之外，每条记录多了一个名叫employees的数组，里面各有两条状态是insert的员工记录。这表示，对第一条记录，在dept表新建一条deptno=3的记录，并在emp表中插入两条属于该部门的新员工记录；对第二条记录，更新deptno=1的dname字段，同时也在emp表中新建两条属于该部门的员工记录。执行后，查看数据库，可见：

```
SQL> select deptno,dname from dept where dname like 'test%';
```

```
DEPTNO DNAME
```

```
-----
      3 test3_created
      1 test1_updated
```

```
SQL> select empno,deptno,ename from emp where ename like 'emp%';
```

```
EMPNO   DEPTNO ENAME
```

```
-----
    8007         3 emp_test_3.1
    8008         3 emp_test_3.2
    8009         1 emp_test_1.1
    8010         1 emp_test_1.2
```


在BM中，通过cascade-operations标记，设置需要进行级联操作的从表，每一个从表设置一条cascade-operation标记，在cascade-operation标记中，通过inputPath属性设置该表对应的子记录集在头记录中的属性名（如上例中的employees），operations属性设置子记录允许的操作，如insert, update, delete, execute等，用逗号分隔。

1. 通过相对路径指定外键字段值

在上例中，员工记录的deptno字段是通过参数指定的。而实际应用中，子记录所对应的父记录主键值，通常来源于父记录。这时，可以在emp表中，设置deptno字段的parameterPath属性：

```
<bm:field name="deptno" parameterPath="../../@deptno" />
```

这表示deptno字段来源于上上层记录的deptno属性。在XPath中，“..”表示当前路径的上一级。由于我们提交的参数结构是每条dept记录下面包含一个employees数组，employees下面包含emp记录，所以对于emp记录来说，向上两级就是它所述的dept记录。

考虑到emp表有可能是单独执行insert，通过参数传递deptno值，也有可能是作为dept的从表在批量操作中执行insert，通过父记录获取deptno值，我们要为两种应用设计不同的BM。后面将介绍BM的继承模式，对于批量操作模式，我们可以从原有的emp中派生一个emp_for_batch_update，其他设置不变，只是改变deptno字段的parameterPath属性。

4.2.4. 自定义BM操作

4.2.4.1. 用自定义SQL实现CRUD操作

对于复杂的业务场景，框架自动创建的SQL如果不能满足业务需求，可通过自定义SQL的方式来替代标准的SQL。和自定义查询类似，自定义insert, update, delete的SQL语句，也是通过在BM的operations标记下面新建对应名字的operation来实现的。例如：

```
<bm:model xmlns:bm="http://www.aurora-framework.org/schema/bm" needAccessControl="false">
  <bm:operations>
    <bm:operation name="insert">
      <bm:parameters>
        <bm:parameter name="new_name" required="true"/>
        <bm:parameter name="source_empno" dataType="java.lang.Long" required="true"/>
        <bm:parameter name="new_empno" dataType="java.lang.Long" output="true" input="false" />
      </bm:parameters>
      <bm:update-sql>
declare
  v_id number;
begin
  select emp_s.nextval into v_id from dual;
  insert into emp t ( empno, ename, job, mgr, hiredate, sal)
  select v_id, {@new_name}, e.job, e.mgr, trunc(sysdate), sal
  from emp e where empno = {@source_empno};
  {@new_empno} := v_id;
end;
      </bm:update-sql>
    </bm:operation>
  </bm:operations>
</bm:model>
```

```

<bm:operation name="delete">
  <bm:parameters>
    <bm:parameter name="emp_id" dataType="java.lang.Long" required="true"/>
  </bm:parameters>
  <bm:update-sql>
    begin
      emp_maintain_pkg.delete_employee(${@emp_id});
    end;
  </bm:update-sql>
</bm:operation>
</bm:operations>
</bm:model>

```

在这个BM中，我们在operations部分，分别定义了两个insert, delete两个操作，并通过下面包含的update-sql标记，定义这两个操作各自对应的SQL语句。这样，对该BM执行insert或delete操作时，就会执行对应的自定义SQL语句。

4.2.4.1.1. 参数定义

对于自定义操作需要的输入或输出参数，必需通过parameters部分进行定义，如上面的insert操作，定义了名为new_name, source_empno, new_empno三个参数，前两个用于输入，第三个用于输出，获取新创建的emp记录id。

如果参数是直接来自于客户端提交的参数，可以通过name属性来定义参数的名称，否则，需要通过inputPath属性定义参数的XPath路径，如inputPath="/session/@user_id"。如果是必需的参数，应设置require="true"。如果参数的数据类型不是字符串，这样，如果客户端没有传递该参数，框架会直接返回错误信息，避免产生脏数据的可能。如果参数的数据类型不是字符串，应通过dataType属性设置实际的数据类型。如果客户端提交的参数格式不正确，框架也会返回错误信息。

4.2.4.1.2. 输出参数的定义

如果参数是用于获取SQL语句的输出值，需要设置属性input="false", output="true"。缺省情况下，输出参数放置在数据容器的/parameter/@[参数名]路径。如果希望放置在其他指定的路径，可以通过outputPath属性设置期望的路径，例如outputPath="/session/@role_id"。

在一个BM中，对于不同的操作，自定义SQL可以和框架自动生成的SQL混用。例如，可以定义fields, primary-key 和 relations，再定义一个名为update的operation，让框架自动生成query, insert的sql，并在执行update操作时使用自定义sql。

4.2.4.2. 存储过程的调用

调用数据库中的存储过程，实现方式也是一样的，只是对于存储过程，我们约定其operation的名称一般为execute。例如：

```

<bm:model xmlns:bm="http://www.aurora-framework.org/schema/bm">
  <bm:operations>
    <bm:operation name="execute">
      <bm:update-sql>

        begin
          sys_notify_pkg.update_notify

```

```

        (
            ${@notify_id},
            ${@description},
            ${@message_title},
            ${@message_content},
            ${@send_method},
            ${@send_mode},
            ${@delay_time},
            ${@delay_time_unit},
            ${@content_sql},
            ${@msg_send_check_proc},
            ${@msg_create_proc},
            ${@enabled_flag},
            ${/session/@user_id}
        );
    end;

</bm:update-sql>
<bm:parameters>
    <bm:parameter name="notify_id"/>
    <bm:parameter name="description"/>
    <bm:parameter name="message_title"/>
    <bm:parameter name="message_content"/>
    <bm:parameter name="send_method"/>
    <bm:parameter name="send_mode"/>
    <bm:parameter name="delay_time"/>
    <bm:parameter name="delay_time_unit"/>
    <bm:parameter name="content_sql"/>
    <bm:parameter name="msg_send_check_proc"/>
    <bm:parameter name="msg_create_proc"/>
    <bm:parameter name="enabled_flag"/>
    <bm:parameter inputpath="/session/@user_id"/>
</bm:parameters>
</bm:operation>
</bm:operations>
</bm:model>

```

4.2.5. 在BM中使用Feature

4.2.5.1. Feature示例: Standard Who

Aurora可以看做是一个面向Feature的框架。所谓Feature，就是指可以重复使用的某种功能或特征。我们先来看一个例子。在企业应用中，经常需要在数据中记录更新信息，例如记录的创建人，创建时间，最后更新人，最后更新时间等。反应到数据库表结构上，就是大多数业务数据表都会有created_by, creation_date, last_updated_by, last_update_date等这样几个字段。每条记录在创建或更新的时候，都应该维护这几个字段。比如，创建记录时设置created_by等于当前登录用户的id，creation_date等于系统日期。

如果用单纯的O/R Mapping的思路去建模，这样特性很难做出合理的可重用设计。如果将这个特性封装在父类中，要求每个有此特性的业务实体都继承指定的父类。对于Java这样的单继承语言来说，一定会遇到某些业务实体必需从其他的父类派生，或者某些业务实体需要从父类派生，但又不需要这种特性的情况。这时，在子类coding就不可避免。对于Hibernate这样的O/R Mapping工具来说，如果出现这种具有一定共性的需求，对于应用开发人员来说，除非对Hibernate的源代码非常熟悉，否则就很难在短时间内扩展Hibernate，去实现这种特性。

Aurora的BM提供了一种基于配置的，声明式的功能扩展机制。以前面的emp.bm为例，我们在配置文件中增加一段feature配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<bm:model xmlns:bm="http://www.aurora-framework.org/schema/bm" xmlns:f="aurora.database.features" alias="e" baseTable="EMP">
  <bm:fields>
    <bm:field name="empno" databaseType="BIGINT" datatype="java.lang.Long" insertExpression="(select max(empno)+1 from EMP)" />
    <bm:field name="employee_name" databaseType="VARCHAR" datatype="java.lang.String" physicalName="ename" />
    <bm:field name="job" databaseType="BIGINT" datatype="java.lang.String" />
    <bm:field name="mgr" databaseType="BIGINT" datatype="java.lang.Long" />
    <bm:field name="hiredate" databaseType="DATE" datatype="java.sql.Date" />
    <bm:field name="deptno" databaseType="BIGINT" datatype="java.lang.Long" />
    <bm:field name="sal" databaseType="FLOAT" datatype="java.lang.Long" updateExpression="trunc(${@sal})" />
    <bm:field name="comm" databaseType="FLOAT" datatype="java.lang.Long" />
    <bm:field name="last_updated_by" databaseType="BIGINT" datatype="java.lang.Long" forInsert="false" forceUpdate="true" />
  </bm:fields>
  <bm:primary-key>
    <bm:pk-field name="empno" />
  </bm:primary-key>
  <bm:relations>
    <bm:relation name="dept" joinType="LEFT OUTER" refModel="test.dept">
      <bm:reference foreignField="deptno" localField="deptno" />
    </bm:relation>
  </bm:relations>
  <bm:ref-fields>
    <bm:ref-field name="department_name" relationName="dept" sourceField="dname" />
  </bm:ref-fields>
  <bm:features>
    <f:standard-who />
  </bm:features>
</bm:model>
```

在BM的最后，声明了features部分，并在其下设置了一个名为standard-who的feature。增加了这个设置以后，再看通过BM生成的insert,update语句，可以看到自动多出了对前面几个追溯字段的处理：

```
INSERT INTO EMP
  ( ..., CREATED_BY, CREATION_DATE, LAST_UPDATED_BY, LAST_UPDATE_DATE)
VALUES
  ( ..., ${/session/@user_id}, sysdate, ${/session/@user_id}, sysdate)
```

```
UPDATE EMP e
SET .... , e.LAST_UPDATED_BY=${/session/@user_id}, e.LAST_UPDATE_DATE=sysdate
WHERE e.empno = ${@empno}
```

如果表中的standard who字段的名称与缺省的不一致，可通过standard-who标记的属性进行配置：

```
<f:standard-who createdByField="create_user_id" creationDateField="create_time" lastUpdatedByField="updated_user_id" lastUpdateDateField="updated_time" />
```

Aurora框架提供了一种机制，可以在配置文件中放入任意xml标记，并将这个标记和某一个或某些java

class关联在一起。进而，这些java class会在框架处理数据的过程中，选择合适的切入点，介入处理过程，对中间数据进行加工、转换，从而实现各种可扩展的特性。对于上面的例子来说，实现standard-who标签的java class，在生成实际SQL之前接管控制，自动在BM中添加4个standard who字段。这样，生成的insert，update语句中，就会多出standard who字段的处理。

对于这个简单的例子，虽然说也可以要求开发人员在自己的BM中都手工加上这四个字段，但有可能会有人失误，少放置了某个字段，或者将某个字段的名称，数据类型拼写错误。以standard-who标签的方式来配置，由于只涉及一个配置点，出错的几率大大减少。更重要的是，这种具有普遍性的业务需求的实现，将集中于一处，更有利于维护。如果今后实现standard who的机制发生改变，例如不是直接存在业务数据表中，而是单独存在一个另外的表，那只需要开发一个新的java class，按新的数据结构去生成SQL语句，并将其与standard-who标签关联在一起就可以，不用去修改已经存在的所有BM。

4.2.5.2. Aurora内建Feature介绍

Aurora框架自带了一些feature，下面将逐一介绍。请注意，虽然看起来这是Aurora框架“内建”的功能，但它们并不是框架的一级成员，而是一种“插件”。它们和用户自行开发的feature完全一样，是通过配置的方式来集成到Aurora框架中的。

4.2.5.2.1. 数据多语言

4.2.5.2.2. Lookup代码

4.2.5.2.3. Oracle sequence

4.2.5.2.4. 标记式删除

4.2.6. BM继承

4.2.6.1. BM继承概述

在Aurora框架中，可通过一个BM配置文件同时实现数据的查询，新增，修改等操作。但是，在实际应用中，往往会碰到这样的情况：执行查询时需要10个字段，并需要join其他的一些表，而执行插入时只需要6个字段；或者，一个BM在大多数查询时都适用，但在某种场合下又要加上额外的限制条件，而查询结果中的字段又是完全相同的。如果为这些应用场景都开发一个单独的BM，势必会有很多重复的成分，相同的配置散布在多个文件里，不仅开发效率低，也会带来维护的麻烦。

Aurora为BM提供了一种继承机制，与java语言中的继承类似，一个BM可以继承自另一个BM，从而自动获取父BM所有或部分配置，并可设置自己特有的属性。

与Java语言中的全盘继承所不同的是，BM可以通过设置继承模式，来有选择地继承父BM中的各种对象。例如，父BM中定义了10个字段，子BM中可以选择只继承其中的3个。下面分别说明BM的两种继承模式。

4.2.6.2. 引用模式

在引用模式中，子BM必须显示地定义哪些对象是希望从父BM中继承过来的。例如：

“test/emp_for_lov.bm”

```
<bm:model xmlns:bm="http://www.aurora-framework.org/schema/bm" alias="t2" extend="test.emp" extendMode="reference">
```

```

<bm:fields>
  <bm:field name="empno"/>
  <bm:field name="employee_name"/>
</bm:fields>
</bm:model>

```

这里，该BM通过`extend="test.emp"`属性，声明从`test.emp`继承而来，并通过`extendMode="reference"`属性，声明继承模式为引用模式。在`fields`部分，定义了两个`field`，并且只设置了`name`属性。这意味着，该BM将从`test.emp`继承`empno`，`employee_name`这两个指定的字段。字段上的属性，如`dataType`，`databaseType`等也将一并继承过来。`test.emp`的其他字段，由于没有在子BM中声明，就不会出现在子BM中。

查看通过该BM生成的查询SQL，可以看到：

```

SELECT t1.empno,t1.ename AS employee_name,dept.dname AS department_name
FROM emp t1
      LEFT OUTER JOIN dept dept ON t1.deptno = dept.deptno

```

由于子BM只声明了两个字段，所以`select`的字段部分只有两个。但是，父BM中定义的`relations`及`ref-fields`依然在生效。下面将详细解释其中的原理。

4.2.6.2.1. 引用模式中集合对象的处理

BM中有很多集合对象，如`fields`是`field`的集合，`ref-fields`是`ref-field`的集合，`query-fields`是`query-field`的集合，等等。在`reference`模式下，对于父BM的集合对象，处理规则如下：

- 如果子BM中对某个集合没有任何设置，则自动从父BM中完全继承此集合。例如，上面的例子中，子BM没有`ref-fields`，`primary-key`和`relations`标记，所以将会从父BM中完全继承这些内容。
- 如果子BM中对某个集合只设置了空的顶层元素，那么子BM中将不会包含父BM中所声明的此集合中的任何元素。例如，在上例中，如果在子BM中放置`<bm:ref-fields />`标记，那么子BM将不包含任何`ref-field`，尽管父BM已经声明过几个`ref-fields`。
- 如果子BM设置了某个集合，并在其中也设置了子节点，那么，将会自动合并父BM在此集合中具有相同`name`属性的节点。如果父BM没有对应`name`属性的节点，那相当于子BM在此设置了一个全新的节点。对于父BM中存在，子BM中不存在的子节点，将不出现在子BM中。

我们再看一个例子，上面的子BM中，我们首先设置`name="empno"`的`physicalName`属性，改为另外一个字段名，再增加一个名为`salary`的`field`，然后在`ref-fields`部分设置一个父BM没有设置过的字段，最后再增加一个`data-filter`：

“test/emp_for_lov2.bm”

```

<bm:model xmlns:bm="http://www.aurora-framework.org/schema/bm" alias="t2" extend="test.emp" extendMode="reference">
  <bm:fields>
    <bm:field name="empno" physicalName="empno1_that_not_exists"/>
    <bm:field name="employee_name"/>
    <bm:field name="salary" expression="trunc(sal)"/>
  </bm:fields>
  <bm:ref-fields>
    <bm:ref-field name="new_dname" relationName="dept" sourceField="dname"/>
  </bm:ref-fields>
</bm:model>

```

```

</bm:ref-fields>
<bm:data-filters>
  <bm:data-filter expression="mrg is not null"/>
</bm:data-filters>
</bm:model>

```

然后查看生成的查询SQL:

```

SELECT t2.empno1_that_not_exists AS empno, t2.ename AS employee_name, trunc(sal) AS salary, dept.dname AS new_dname
FROM EMP t2
      LEFT OUTER JOIN dept dept ON t2.deptno = dept.deptno
WHERE mrg is not null

```

由于empno字段的physicalName属性在子BM中被重新定义，所以生成的SQL中以子BM中设置的字段名为准。salary字段是子BM新定义的，父BM中没有该字段，也会出现在生成的SQL中。由于ref-fields集合在子类中被设置，所以会只出现子BM设置了的ref-fields；而relations集合没有设置，所以relations部分完全继承父BM。由于子BM额外定义了data-filters，所以生成的SQL中多了一个where条件。

4.2.6.3. 重载模式

在BM中，设置extendMode="override"来设置继承模式为重载模式。这种模式与Java语言的继承模式类似，子BM将继承父BM的所有设置，子BM设置的同名属性、节点将覆盖父BM的设置。实例：

“test/emp_for_lov3.bm”

```

<bm:model xmlns:bm="http://www.aurora-framework.org/schema/bm" alias="t2" extend="test.emp" extendMode="override" needAccess="true">
  <bm:fields>
    <bm:field name="empno" physicalName="empno1_that_not_exists"/>
    <bm:field name="employee_name"/>
  </bm:fields>
  <bm:ref-fields>
    <bm:ref-field name="new_dname" relationName="dept" sourceField="dname"/>
  </bm:ref-fields>
  <bm:data-filters>
    <bm:data-filter expression="mrg is not null"/>
  </bm:data-filters>
</bm:model>

```

除了extendMode属性，其他与emp_for_lov2完全一致。对应的查询SQL:

```

SELECT t2.empno1_that_not_exists AS empno, t2.ename AS employee_name, trunc(sal) AS salary, t2.job, t2.mgr, t2.hiredate, t2.deptname AS new_dname
FROM EMP t2
      LEFT OUTER JOIN dept dept ON t2.deptno = dept.deptno
WHERE mrg is not null

```

由于override模式下，所有设置都会从父BM继承，所以生成的SQL中包含test.emp定义过的所有字段。由于子BM对empno字段重新设置了physicalName属性，所以该字段对应的SQL以子BM为准。与前例相同，在子BM中添加的设置，如名为salary的field，以及data-filter，都会生效。子BM定义的ref-field

，由于名称不一样，所以会与父BM定义的ref-field合并，同时出现在SQL中。

4.2.6.3.1. 重载模式中集合对象的处理

在override模式下，对于父BM的集合对象，处理规则如下：

- 如果子BM中对某个集合没有任何设置，则自动从父BM中完全继承此集合，这与reference模式相同。
- 如果子BM中对某个集合只设置了空的顶层元素，如<bm:ref-fields />，那么子BM一样会继承父BM对该集合的所有设置，效果等同于没有设置这个顶层元素。
- 如果子BM设置了某个集合，并在其中也设置了子节点，那么，将会自动合并父BM在此集合中具有相同name属性的节点。如果父BM没有对应name属性的节点，那相当于子BM在此设置了一个全新的节点。对于父BM存在而子BM没有定义的节点，也会被子BM所继承。

4.2.6.4. 同名节点的属性处理

所谓同名节点，是子BM和父BM中，同一集合下，具有相同name属性的节点，例如fields下面中具有相同name的field。如果父节点和子节点都设置了相同的属性，那么以子节点的为准，如前面例子中，empno字段中的physicalName属性。如果父节点设置了某属性而子节点没有设置，则自动从父节点继承此属性。如果子节点设置了而父节点没有设置，一样以子节点为准。

这些规则同样也适用于BM的根节点：model。例如，父BM中设置了<bm:model alias="t1" >，子BM中设置了<bm:model alias="t2" >，那么最终的alias将是t2。

4.2.6.5. BM继承应用场合分析

以下就各种典型的应用场合，分析子BM应该如何设置。

- 需要添加额外的字段：用override模式，在子BM中只定义要加的字段。
- 需要更改字段的属性：如果父BM中所有的字段都要用，那么用override模式，否则用reference模式，在fields部分逐一设置需要继承的field及其name。然后，在fields中设置新的属性值。
- 需要减少字段：用override模式，否则用reference模式，在fields部分逐一设置需要继承的field及其name。
- 需要增加额外的查询限制条件，或可选查询条件：先根据fields部分的需求，确定用那种模式。对于override模式，直接在data-filters或query-fields部分增加新的data-filter，query-field即可。对于reference模式，如果父BM的data-filter或query-field是需要继承的，那么还需要逐一设置需要继承的data-filter或query-field子节点及其name属性。
- 需要取消父BM中的某个限制条件：首先，父BM中的每个data-filter都应该用name属性命名，以便引用。如果是reference模式，在data-filters部分不设置父BM中对应的这个data-filter节点就可以了。如果是override模式，可以在子BM的data-filters下面设置同名data-filter，并将其expression属性设置为true（或等同于true的表达式，如1=1），效果相当于没有这个限制条件。
- 需要取消父BM中关联的表，字段：用reference模式，然后在relations及ref-fields部分，不设置要取消的关系及字段。

- 需要关联新的表，字段：不论用那种模式，在子BM的relations及ref-fields部分增加新的relation及ref-field，去关联新的表。
- 需要更改某个relation所关联的BM名称或join type：不论用那种模式，在子BM的relations下面设置同名relation节点，并设置新的refModel或joinType属性。